KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering

Decentralized Service Selection and Composition

Cruz Torres Mario Henrique

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

January 2015

Decentralized Service Selection and Composition

Cruz Torres MARIO HENRIQUE

Supervisory Committee:
Prof. dr. ir. Joos Vandewalle, chair
Prof. dr.Tom Holvoet, supervisor
Prof. dr. Danny Hughes
Prof. dr. ir. Wouter Joosen
Prof. dr. Paul Valckenaers
Prof. dr. Filip De Turck
(Universiteit Gent)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

January 2015

Uitgegeven in eigen beheer, Cruz Torres Mario Henrique, Celestijnenlaan 200A box 2402, B-3001 Heverlee (Belgium)

ISBN XXX-XX-XXXX-XXX-X

D/XXXX/XXXX/XX

Abstract

Services computing facilitates the creation of large scale applications. Services are relatively small and manageable software units with clearly defined interfaces. Applications then consist of orchestrated invocations of services, the so-called composite services. The services on which a composite service relies - called component services - have various quality of service (QoS) characteristics, such as performance, reliability, availability, accuracy. Such quality parameters can be used by a composite service to select component services when called for. Service selection and composition is particularly challenging when the system is large-scale - consisting of thousands of nodes, components and composite services - and dynamic - where QoS varies.

Particularly challenging is the situation where potentially large failures can occur. This thesis ambition is to create a highly resilient system for dynamic service compositions. While traditionally, a resilient system is able to deal with failures, our aim is to conceive a system that considers failures as 'business as usual', to which it gracefully molds itself.

The large or potentially huge scale of such systems (involving tens of thousands of nodes and services) makes a central selection and composition authority infeasible. In our research, we investigate a decentralized self-adaptive and self-organizing approach to dynamic service composition. In particular, we study delegateMAS, a coordination mechanism originally targeted for large-scale coordination and control applications, such as traffic and logistics management, where entities need to coordinate over resources. Such coordination and control systems are intrinsically dynamic due to changes in operational (uncertainty of service time, orders, travel demand) or exceptional conditions (vehicle failures, infrastructure problems).

In this thesis we, first, define a decentralized solution for dynamic service composition using delegate MAS. We define TaskAgents that are responsible for enacting composite service instances, and ResourceAgents that manage the

usage of component services. Two delegate MASs are defined, for exploring compositions and for propagating information about intended compositions.

Second, we thoroughly investigate the scaling of the system. We ran experiments that were large and huge in scale (up to tens of thousands of nodes and services). Third, we assess the behavior of the system under failing conditions, including drastic failure scenarios. These experiments show that the approach is effective, efficient, scales linearly, and can cope even with severe failures.

We validate our approach by applying it to different domains and by performing a thorough evaluation of it. We conclude this thesis showing that it is possible to create service compositions which can cope with failures, without relying on centralized solutions.

Beknopte samenvatting

Het gebruik van services maakt het ontwikkelen van grootschalige applicaties eenvoudiger. Services zijn kleine makkelijk te beheren softwareonderdelen met duidelijke interfaces. Applicaties worden dan samengesteld uit gecoördineerde aanroepingen van die services, samengestelde services genaamd. De services waarop zulke samengestelde services verder bouwen - component services - hebben verschillende kwaliteitseigenschappen zoals performantie, beschikbaarheid, betrouwbaarheid en accuraatheid. Deze kwaliteiten bepalen de selectie van component services bij een oproep van een samengestelde service. De keuze van component services is een grote uitdaging wanneer het systeem groot is - met duizenden component en samengestelde services - en dynamisch - met verschillende kwaliteitseigenschappen.

Het keuzeprobleem is vooral moeilijk wanneer veel services kunnen falen. Het doel van deze thesis is om een erg betrouwbaar en veerkrachtig systeem te ontwikkelen voor dynamische servicecompositie. Betrouwbare systemen kunnen typisch goed omgaan met het falen van services. Ons doel is een systeem te ontwikkelen waar het falen van services als gewoon beschouwd wordt.

De grote schaal van zulke systemen, met tienduizenden nodes en services, maakt een centrale selectie en sturing van de compositie onmogelijk. In ons werk onderzoeken we een gecentraliseerde zelf-aanpassende en zelf-organiserende aanpak voor dynamische servicecompositie. We bestuderen delegateMAS, een coördinatiemechanisme origineel bestemd voor grootschalige coördinatie en controletoepassingen zoals logistiek en verkeer. Coördinatie in zulke systemen zijn van nature dynamisch omwille van wijzigingen in operationele (onzekerheid van reistijden, orders of vraag) of uitzonderlijke omstandigheden (falen van voertuigen, problemen met de infrastructuur).

In deze thesis definiëren we eerst een gedecentraliseerde oplossing voor dynamische servicecompositie met delegate MAS. We definiëren TaskAgents die verantwoordelijk zijn voor het uitvoeren van de samengestelde services,

en ResourceAgents die het gebruik van component services beheren. Twee delegate MAS'en zijn gedefinieerd, voor het verkennen van de composities en voor het verspreiden van informatie over de mogelijke composities. Daarna onderzoeken we de schaalbaarheid van het systeem. We voerden grootschalige experimenten uit met tot tienduizenden netwerkknopen en netwerkservices. Ten slotte bekijken we het gedrag van het systeem wanneer er veel services falen. We bekijken ook scenario's waarin er erg veel services falen. De

experimenten tonen aan dat onze aanpak werkt, efficient is, lineair schaalt en meerdere falingen aankan.

We valideren onze aanpak door hem toe te passen op verschillende domeinen en door hem grondig te evalueren. We besluiten deze thesis door aan te tonen dat het mogelijk is services samen te stellen zelfs wanneer er services falen zonder de nood aan centrale oplossingen.

Acknowledgments

Anything is created with the direct, or indirect, support of other people, and this thesis is no different. This thesis wouldn't exist if it wasn't for the continuous support and encouragement of many people that somehow shared this doctorate journey with me.

I am very grateful for the time I spent at the 04.120 office, where I could concentrate on my work and also have very interesting conversations. I am glad for my office mates Rutger Claes, Robrecht Haesevoets, Koen Buyens, Alexander Helleboogh, Tom Govaerts, Fan Yang, Wilfried Daniels, Rafael Bachiller Soler, and Eduardo Cañete Carmona. They never expelled me from the office, even with my not so conventional approaches to work, like my pomodoros, office fruits, and my love for the white board. I am also glad by never having realized our plans of reorganizing our office. That showed me I wasn't the only lazy one in the office.

I am indebted to my colleagues at iMinds-Distrinet with whom I could share my frustrations and also my accomplishments. Hopefully not forgetting any names, I would like to thank Jan Tobias, Ilya Sergey, Dimitar Shterionov, Dimiter Milushev, Rinde Van Lon, Fatih Gey, Gowri Sankar Ramachandran, Stijn Vandael, Rula Sayaf, Aram Hovsepyan, José Proença, and Bartosz Michalik. With many of these colleagues, I only had occasional conversations while queuing for the coffee, but these smalltalks were invaluable, nonetheless.

After spending many years at the 04.120 office I was changed to a new office, in front of the old one, where I again found amazing colleagues, as Caren Crowley, Stefan Walraven, Bart Vanbrabant, Wouter De Borger.

It is impossible to talk about iMinds-Distrinet and not to mention the support from Marleen Somers, Katrien Janssens, Ghita Saevels, Esther Renson, and Annick Vandijk who always had a way to help. Either helping me to organize trips to conferences, giving ideas to better poster designs, demos, or having a nice word at the coffee queue.

Life outside academia and far from family wouldn't be bearable if it weren't the nice moments I had with friends as Hugo Sobreira, Heverly Herr Sobreira, Jerry and Corina Beck, Analu and Kris Waugh. I also want to thank my family, for always being supportive and loving me while living for so many years far from them. Thank you my brothers Dalmo and Daniel for the few, but very nice moments we passed together in the last years. My father was always excited to hear my comments about the life abroad and what I was learning about the world. Thank you Dad. I can not forget to thank my mother Maria Emilia, as well, whom was always enthusiastic about life and whom thought me so much. I also like to thank my in-laws Ismênia Felix Rigolin and Antônio Roberto Rigolin, whom, even being in-laws, were always supportive, taking this very long flight to visit my family in Belgium.

Last but not least, I thank my wife Daniele Cristina Rigolin and my daughters Joana, Cecilia, and Isabela, for being the reason of my life, sharing my dreams, believing in me, and giving me a reason to open my eyes every day. I love you.

Living abroad comes with highs and downs and with a constant learning. I thank my adviser Tom Holvoet for accepting me in this journey and offering a good advice at times I needed. I will never forget the opportunity you gave me... to learn about myself, my own culture, and distributed systems.

Abbreviations

2PC	Two Phase Commit Protocol
ACO	Ant Colony Optimization
APFL	Adaptable Pervasive Flow Language
BDI	Belief-Desire-Intention
BPEL	Business Process Execution Language
CASAS	Composable Adaptive Service Agent System
Delegate MAS	Delegate Multiagent System
DoS	Denial-of-Service
EAI	Enterprise Application Integration
ESB	Enterprise Service Bus
HTTP	Hyper-Text Transfer Protocol
MAS	Multiagent System
NTP	Network Time Protocol
QoS	Quality of Service
REST	REpresentational State Transfer
SLA	Service Level Agreement
SLO	Service Level Objective
SOAP	Simple Object Access Protocol

UDDI	UDDI
URI	Uniform Resource Identifier
WS	Web-service
WS-*	Web-Service Protocols
WSDL	Web-Services Description Language
XML	Extensible Markup Language

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Creating large scale distributed applications is difficult due to different factors such as increased complexity, uncertainties of the networked environment, and computer nodes failures. Services computing facilitates the creation of loosely coupled applications that are spread over a network. Even more, services computing allows the integration of applications belonging to different organizations. Such integration is only possible thanks to the use of standardized protocols that are widely used and accepted by several organizations.

In Services computing, systems are decomposed into smaller and more manageable units called services. Services are software entities with well defined operations that can be invoked via the network. Services computing allows the creation of applications that are, normally, easier to evolve, maintain, and can better cope with the particularities of the network environment. Because services use standardized protocols for communication and data exchange, it is possible to implement service clients and servers using any desired programming language.

Decomposing systems into services enables the use of such services in ways that were not foreseen at the time of their creation. Another benefit is that applications belonging to one organization can use specialized services from other organizations, letting those institutions focus on their core expertise.

Simply having a large number of services, however, does not constitute any useful functionality of an application. Services need to be invoked in a particular way to provide meaningful behaviour to an application. Their execution needs to be orchestrated or they need to coordinate their execution according to a set of rules.

The main idea behind service orchestration is that, many times, it is possible to reuse existing services in order to create new applications. It is also possible to expose the newly created applications as new services, called composite services.

Composite services, in turn, orchestrate the execution of a number of services in order to achieve their objectives. The services on which a composite service relies may have different quality parameters, for instance, a particular service may offer fast computations but no reliability while another service may offer moderate speeds for computations but high reliability. Quality parameters, known as quality-of-service parameters (QoS), can be used by a composite service to deciding which are the best services to use at a certain moment. However, deciding which are the best services is computationally expensive, specially because of the number of services and the many quality parameters such as availability, efficiency, correctness, price, and response time.

Another characteristic of composite services is that they can be bound to component services, which provide the needed operations, either statically or dynamically. On the one hand, statically bound composite services provide certainty regarding which component services will provide the needed operations. On the other hand, statically bound composite services are inflexible in the event of failures or QoS changes. Dynamically bound composite services select their component services at runtime, either from a set of known services or by other means, such as querying service registries. That way, dynamically bound composite services are flexible in the event of failures or changes in the operating environment.

1.1 Problem Statement

Our research is about dynamically binding composite services, which has many challenges. The first challenge concerns the scale of the system regarding the number of component services that can possibly be used by a composite service. In other words, assuming there are thousands of replica component services, each one having a different quality, selecting which services should be used by a composite service at a particular time is computationally challenging.

Dynamic events are another source of complexity in composite services, for example, the quality of a service can degrade, new services may become available, the number of composite services operating in the system may have sudden increases, network links may stop working changing the network topology, new service versions may be deployed and co-exist with older service versions, etc.

The QoS of composite services, in our study those participating in a large scale

application, can vary a lot depending on the services they use. Even leveraging the existence of several replica services it is challenging to guarantee a certain composite service's QoS. Thus, a second challenge is to guarantee a given QoS of a composite service in an open network environment where service applications are executed.

Failures constantly happen in large scale service systems. There are different sources of problems that can cause failure, such as faulty hardware, network failures, server misconfiguration, etc. Such failures may directly have consequences on services as well, for instance services may become inaccessible at a time and later become accessible again, or services which are already participating in a service composition may stop processing their operations, etc.

Furthermore, it is particularly challenging to cope with large scale failures. Large scale failures affect a large proportion of services in a system and may have enormous consequences, such as halting the entire system.

Ideally, the composite services in a large scale service system should not stop working when a number of component services fail. However, coping with a large number of component services, selecting the best quality services, and dealing with failure at the same time is challenging. A third challenge concerns diminishing the effects of system failures on composite services execution.

A naive approach to monitor large service systems is to constantly monitor each component service by polling their current QoS parameters. Such approaches only allow applications to reactively handle failures or QoS changes. Constantly monitoring all component services may be suited for small-scale systems. This approach may also suit systems which normally only have minor disturbances and limited, or known loads.

However, only monitoring the current state of the component services is not enough to create large scale and highly dynamic service applications. The problem of state-of-the-art techniques is that they would yield unacceptable network overhead, increased servers load, and the duplication of QoS computation. Non-coordinated applications monitoring and reallocating their resources can lead to chaos, for instance, since multiple applications can decide to rebind to the same remaining services, overloading them.

Our research aims at providing building blocks to understanding and creating decentralized large scale service applications, which face the challenges explained above. Below we summarise the main problems we study in this thesis:

- Service usage is not coordinated. Not coordinating service re-bindings in very large services systems and under large scale failures can lead to chaotic situations and even lead to the complete halt of the system.

- Independent QoS retrieval. QoS Information is independently gathered by different composite services. Independently monitoring the set of available services increases the load on the network and on the component services.
- Current state monitoring. Monitoring the current state of several services leads to a system that can at most simply react to changes after they have occurred. The system as a whole does not use available information to better use the available resources.

As explained in this section, there are many challenges to be overcome in order to create large scale service applications. Services working as part of a large scale service application need to coordinate their activities which leads to our research objectives.

1.2 Objectives

Our high level research objective is to enable the coordination of thousands of composite services operating in the same system. To achieve this objective we intend to investigate different aspects concerning the problems of creating large scale service systems.

Firstly, we aim at designing decentralized mechanisms that allow the creation of applications that can easily cope with: i) the scale of the system, in the number of services; ii) dynamic changes in the operating environment; iii) continuous failures.

Secondly, we intend to provide such mechanisms as a middleware that can facilitate the creation of completely decentralized service applications. Noting that our middleware should not rely on any central servers, or nodes, which have privileged knowledge about the structure of the system or its resources.

Thirdly, we also aim at studying the problem of service selection and composition and learn which are the main problems pursued in the research field. Our ambition is to provide abstractions to designers of large scale service systems, as well as to facilitate the creation and understanding of such systems.

Our fourth objective is to study how MAS techniques can be applied to the service selection and composition problem since the MAS community has a track record of studying decentralized systems.

The fifth objective of ours is to perform thorough evaluations of our mechanisms, via large scale emulations, so that we can better assess their quality.

We mainly evaluate our mechanism by focusing on three qualities, which are: i) composition time, ii) communication costs, iii) fairness in resource allocation. The **composition time** is used to evaluate the efficiency of the mechanism in selecting the best services at a particular time. **Communication costs** are used to measure the communication overhead of the mechanisms. We are also interested in the overall allocation of services to composite services, which is measure as the **fairness** achieved by the selection mechanism.

Finally, we also aim at investigating the behaviour of our mechanism under massive failure because large scale service systems have to be able to handle failures from component services in a smooth manner. As mentioned in Section 6.2.2, many types of failure can happen in the system, for instance, a failure can happen at a particular component service implementation, at the computer where the service is deployed, at the datacenter where a number of services are deployed, on the network, etc. We focus on massive failure, by which we mean that a large percentage for instance 20% or 80% of the services of a service system will be in a failed state at any time.

1.3 Contributions

We applied the concepts of our coordination mechanism to a large scale system having very large failures. We showed how it is possible to create resilient service compositions even in the presence of such failures.

- We presented this contribution in the paper entitled “Self-adaptive Resilient Service Composition” published in the proceedings of IEEE ICCAC (2nd IEEE Conference on Cloud and Autonomic Computing), 2014.

We designed a service selection mechanism that can be used in conjunction with existing services in order to enable the decentralized creation of composite services. The selection of each component service was done via run-time monitoring each service’s QoS parameters. This mechanism was created in the form of a middleware that could easily be used by composite service engines.

- We presented this contribution in the paper entitled “Composite service adaptation: a QoS-driven approach” published in the proceedings of Comsware (5th International Conference on Communication System Software and Middleware), 2011.

We applied MAS organizational concepts so as to facilitate the dynamic re-binding of composite services at runtime, using standard WS-* technologies. The focus was to identify abstractions that could be used to facilitate the creation of rules for re-binding component services in order to satisfy any desired service level agreements needed by an application.

- We presented this contribution in the paper entitled "MAS organisations to adapt your composite service" published in the proceedings of MONA+ (3rd International Workshop on Monitoring, Adaptation and Beyond), which was held together with ECOWS (8th IEEE European Conference on Web Services), 2010.

We used our techniques to allow the execution of computing intensive applications on mobile phones. Mobile phones have an increasing computing capabilities, however using such capabilities drains their small batteries. We created a middleware capable of off-loading computations from the mobile phone to services running on the cloud back-end. Our contribution was to coordinate the loading of virtual machines on a cloud provider, so as to minimize the latency perceived by mobile phone users and to minimize the costs of using the cloud provider.

- We presented this contribution in the paper entitled "Resource allocation for cloud-assisted mobile applications" published in the proceedings of IEEE Cloud (5th IEEE International Conference on Cloud Computing), 2012.

A further contribution was related to the application of our techniques to another domain. Instead of directly coordinating services, we applied our techniques to coordinate mobile phone users, seen as resources by the system. The problem explored in this contribution was that mobile phone users had to perform tasks which can not be performed by a single user. We solved this coordination problem by modeling the user tasks as composite services, which needed the operations of other users, which were modeled as component services, and were thus coordinated. We created a middleware which frees applications from directly managing the interactions between users by offering a coordination mechanism to applications. The middleware also used contextual information so as to dynamically determine the best users to execute tasks in the system.

- We presented this contribution in the paper entitled "CooS: coordination support for mobile collaborative applications" published in the proceedings of Mobiquitous (International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services), 2012.

We applied the ideas of Delegate MAS to the supply chain domain. Assuming that each partner in a supply chain exposed a number of services, a task in the supply chain had to be handled by the different partners (services). We studied how to minimize the variance in the time in order to complete the tasks in a supply chain. This work showed an initial version of the concepts that we use in our coordination mechanism.

- We presented this contribution in the paper entitled "Towards robust service workflows: a decentralized approach" published in the proceedings of CoopIS (19th Cooperative Information Systems), 2011.

We also identified similarities between the techniques we used to coordinate the actions of different services and other MAS coordination techniques applied to other domains, such as traffic coordination applications. We extracted the similarities of these coordination techniques and documented them as a design pattern.

- We presented this contribution in the paper entitled "(No) More design patterns for multi-agent systems" published in the proceedings of AGERE!, (1th International Workshop on Programming based on Actors, Agents, and Decentralized Control), 2011.

1.4 Organization of this Thesis

This thesis is organized as follows.

Chapter 2 provides background information on services computing, MAS coordination, overlay networks, and gossiping protocols, which are needed to understand our approach.

We define a system model, in Chapter 3, in order to highlight our assumptions. We present how we conceive the network, the services, and the failures we will work with.

Chapter 4 describes our approach to create decentralized service compositions. In this chapter we discuss in detail how our algorithms work and which abstractions we use to create them.

In Chapter 5 we present a throughout evaluation regarding different aspects of our approach. We test three hypothesis concerning large scale service applications and which allow to understand the behaviour of a system using our algorithms.

In Chapter 6, we show a validation of our research via two papers, published in different venues, which explore different aspects of our research.

We present the state-of-the-art literature from our research field in Chapter 7. We conclude this thesis in Chapter 8. We present our conclusions, lessons learned, and directions for future research.

Chapter 2

Background

In this thesis, we present a novel decentralized service selection and composition mechanism. Our mechanism draws inspiration from different fields. We use techniques from Web-services, Multiagent Systems, and Overlay networks. This chapter introduces each of the above mentioned topics in order to facilitate the understanding of our mechanism.

Our mechanism operates on several software entities, which are distributed on separate nodes in a computer network. The communication between these software entities benefits from the communication facilities provided by web-services. The research presented in this thesis revolves around autonomous software entities, the agents, that communicate in order to achieve a global system optimization. We are not interested only in a single autonomous software entity, but on the behaviour of a system made of several autonomous entities. Multiagent Systems provide our research with useful abstractions to decompose our system into understandable units. We also use techniques from overlay networks and gossiping protocols which provide mechanisms to lower communication overhead and improve the system resilience to failures.

2.1 Web-Services

Web-services were the result of efforts to allow the creation of software that would integrate cross-organizational systems [3]. Interoperability was, in fact still is, a major issue for integrating large scale systems. Different organizations use different software technologies to build their systems. Each technology

dictates how information is processed, stored, and transferred. Additionally it was common to have proprietary protocols for transferring information from one system to another, leading to big efforts to interconnect a system to another system built on top of another technology stack.

Middleware solutions, like Message Brokers and Workflow Management Systems were proposed to solve the integration issues, however they were also not fit to completely solve such issues. For instance, EAI interactions are short lived, while many cross-organizational interactions are in fact long-lived. Protocols, like 2PC, commonly used in EAI and other middleware platforms are not suited for long lasting operations, due to their possibility of locking resources on the applications participating on the interaction.

Another aspect of why conventional middleware technologies were not suited to integrate applications on different organizations lays at the organizational level. Conventional middleware solutions had to be adopted by all organizations involved in the integration effort. All organizations had to trust the same middleware, and many times, the same middleware vendor. An alternative was to have the middleware running on a third party trusted organization, that would be responsible for managing the interaction flows between the organizations [4]. However, most organizations were not inclined to accept solutions that could potentially diminish their autonomy, or that could harm the organization's transactions.

Point-to-point integrations, on the other hand, lead to increased complexity in maintaining the applications, since new integration efforts had to be done for each new collaborating partner. For instance, two organizations could agree to use a particular message broker technology, while a third organization could require to use another message broker.

Due to the problems illustrated above, web-services were mainly created to solve the integration issues of applications from different organizations. The most crucial aspect of web-services was the creation of standards and techniques aiming at creating systems that could easily cross organization boundaries.

The main idea of a web-service is that organizations can easily expose functionality, via the internet, to applications belonging to either internal or external organizations. A web-service is a software that provides a well defined interface and is accessed over the network. A system can be decomposed in a number of services, with each service providing a particular functionality required by the system. These services can be discovered and accessed via the internet by other systems which also adopt the same standards.

There are mainly two categories of web-services, which differ not only in their technology stack, but also in the reasoning on how to decompose and architect

a large system composed by services. The first category, normally called WS-*, heavily relies on XML based standards, which describe communication protocols, interfaces, and data encapsulation methods [80]. The second main category of web-services, called `RESTFul` web-services , or simply REST services, mainly rely on the four basic operations present on the HTTP protocol to provide a uniform interface to the services.

2.1.1 WS-*

The WS-* stack protocols describe how web-services are exposed on the internet and how data is transferred. The standardization process resulted on a set of protocols, like WSDL, `Web Services Conversation Language (WSCL)`, SOAP, `WS-Addressing, WS-Notification, UDDI`, etc. For our purposes it is more important to know the concepts behind the main protocols than their particularities.

The WS-* approach has protocols for finding web-services, binding to them, and requesting operations on the selected services. It also defines protocols for security, coordination and composition of services. The main idea of WS-* services is that there are three roles on a typical service scenario. There is a service requestor, a service provider and a service repository. The `Service Requestor` is the application requesting a service, the `Service Provider` is the application providing services to other applications or services. When a service needs to invoke an operation on a service provider, the service requestor can first lookup the provider on a service registry, and then invoke the operation on the service provider. The service registry is defined at the UDDI protocol. A typical WS-* services scenario is described on Figure 2.1

A WS-* service, normally, provides a set of operations that can be grouped logically. For instance, an often used way to design a service is to represent the functionality of a sub-module of a system as a service. We describe a typical web-service for a banking system at Listing 1.

Algorithm 1 A typical ws-* service to manipulate a bank account.

```
procedure BankAccountService
end procedure
procedure withdraw(Real amount)
end procedure
procedure deposit(Real amount)
end procedure
```

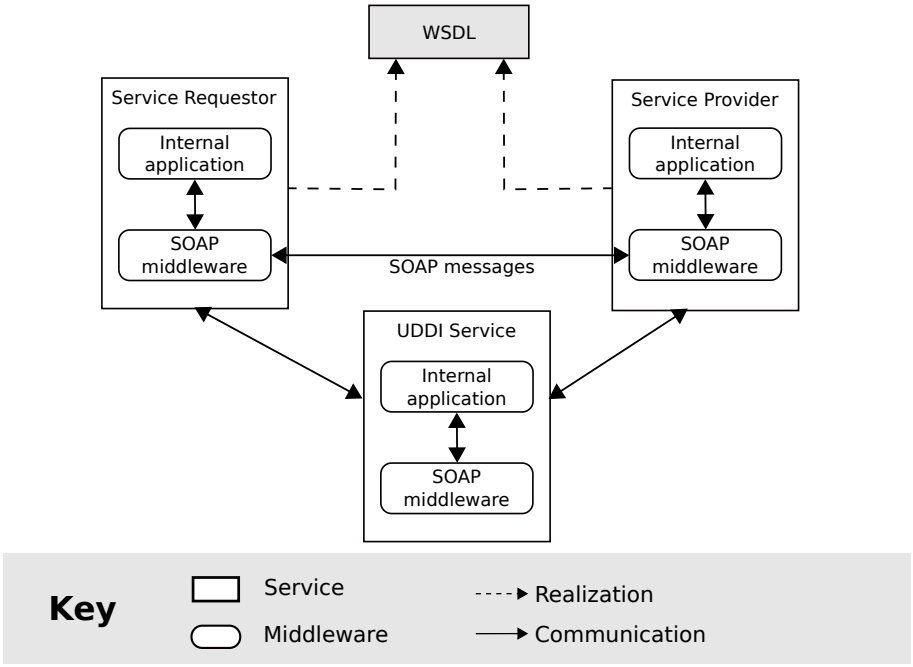


Figure 2.1: Typical roles of WS-* services. A `Service Requestor` and an `Service Provider` agree on a common service interface described in `wsdl`. A `Service Requestor` can find `Service Providers` on a `UDDI` service registry.

The standard way to describe the operations offered by a WS-* service is using WSDL. A WSDL interface describes which operations are provided by a web-service and which data formats it accepts. The data-formats also need to be standardized, to guarantee that all parties involved on a service execution understand the data. A WSDL specification is divided in two parts, an abstract and a concrete part. Definitions of the types used, message formats, and operations are specified on the abstract part of the WSDL file. The concrete part contains information about bindings, like message encodings and protocol bindings for the operations defined in a given port. The concrete part also contains information about the EndPoints, which is the combination of bindings with particular network addresses (specified as URI).

The data transferred between a service requestor and a service provider is encapsulated using SOAP, which is a XML language to describe data formats.

Knowing the WSDL interface, the bindings, and the URI of a service provider, a client can then invoke operations on this provider. Remembering that a client

can not enforce the execution of the operations of a service provider, but only request such execution.

The main critique to the WS-* set of standards is that they became extremely complex, leading to different implementations not being able to properly intercommunicate. Besides the intercommunication problem, WS-* services are hard to test and debug, mainly because of complexity of the middleware used to encapsulate messages, to start a remote invocation, etc. Lastly, WS-* middleware vendors pushed solutions as ESB which lead to centralized pieces of software which are hard to maintain and to scale for higher transaction numbers.

2.1.2 REpresentational State Transfer (REST)

The REST architecture provides another approach to decompose a system into fine grained services. REST services have a standardized unified interface via four basic operations, which are verbs of the HTTP protocol (GET, POST, PUT, DELETE). Unlike WS-* services, a REST service reflects a system decomposition in terms of the system's resources. One has to carefully identify which resources are part of a system, since the only operations allowed to be applied to a resource are always the same basic operations [44].

A REST service follows four basic architectural principles, which are detailed below:

- Resource identification through URI
- Uniform interface
- Self-descriptive message
- Hypermedia as the engine of application state

A `RESTFul` service exposes resources using a URI as a fine-grained interface mechanism. That way, it is possible for another service to directly request information about a particular resource, or to apply operations on it.

To manipulate resources of a `RESTFul` service, one has to use a uniform interface. The uniform interface provides operations to: 1) retrieve a representation of a resource, using a `GET` operation; 2) create a new resource, using the `POST` operation; 3) update the state of a resource, using the `PUT` operation; 4) delete the state of a resource using the `DELETE` operation [80].

The communication between a client and a service is done via messages, which have to be self-descriptive. A self-descriptive message contains information

about its data and meta-data that describes both the format and meaning of this data.

All the interactions with a resource are stateless. The application changes from one state to another via the different hyperlinks which constitute the application.

As an example, a banking application could have the following service:

`Account Service`

- GET, retrieves the account balance
- PUT is used to add a certain amount of money to the account
- DELETE value, withdraws value from the account

REST services are simple to use, mainly because they leverage well known standards such as `HTTP, URI, MIME`. Additionally, the messages are easier to understand and the middleware to support REST services is very lightweight. Even more, REST services very closely resemble the structure of the network, and leverage from such structure as well, lowering the gap between application architecture and the deployment of such applications.

2.2 Composite Web-services

One of the main benefits of creating systems using web-services is the possibility to compose several fine grained services into more complex ones, called composite services. Composite services facilitate restructuring complex service systems according to changes on business processes and business goals [33].

A composite service is made by a number of activities which are delegated to other services. A composite service orchestrates the execution of a number of component services, by invoking the component service's operations at a particular time and order. Conceptually, a composite service can be seen as a graph containing the activities which need to be performed as the nodes of the graph and the data flow as the edges.

Constructing composite services can be challenging both technically and business wise. A business has to know and document all its business operations and provide them as services that can be composed. Figure 2.2 depicts a simple composite service that could be used to handle purchase orders.

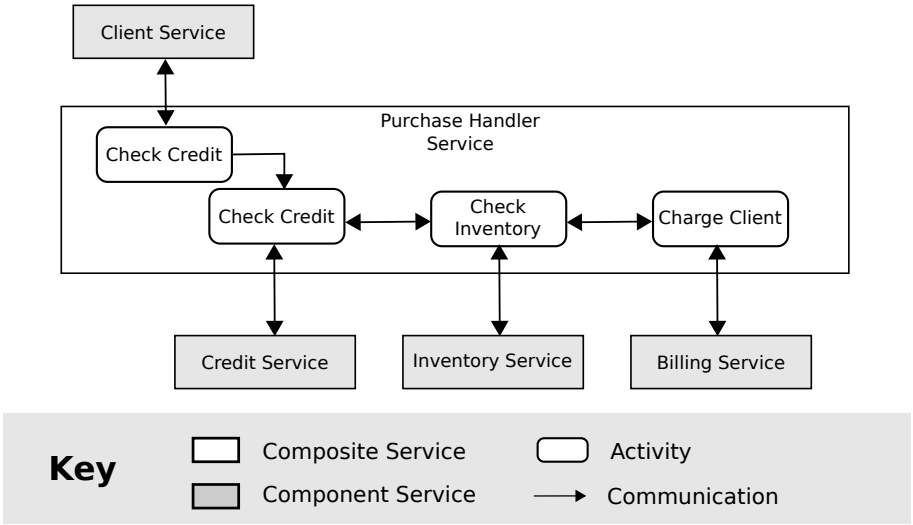


Figure 2.2: A typical composite service could be created to handle purchase orders from clients. Initially a Client service invokes the composite service, placing an order. The composite service has to check the credit information from the client, then check the company inventory and conclude the operations by charging the client and informing the client service if the purchase was completed or not. Note that all operations are delegated to specialized component services.

Creating a composite service is technically difficult due to the distributed nature of service systems and the asynchronous nature of the problem.

A particular challenging aspect of designing composite services is to guarantee that the composite service presents the expected behaviour. There is an entire research around the correctness of service composition, which uses several tools, from static analysis to model checking [54].

A technical difficulty is also on how to express the service compositions and how the compositions should change in the event of failures, or changes in the environment. A common approach is to create a monitoring layer, which is responsible for monitoring the component services which are participating in a service composition [70, 10]. The monitoring layer can trigger events at the composition layer, enacting an adaptation of the service composition.

We focus on the enacting of the service composition, not the design particularities of such.

There are different languages to create service compositions, such as BPEL,

JOpera [7, 79]. A composition language can facilitate the development of composite services via providing integrated visual environments where a designer can graphically create the desired compositions, as in JOpera. Normally, a composition language is executed on a middleware runtime responsible for performing the remote invocations of other services and managing messages requests/response cycles [26].

2.3 Multiagent Systems

There are multiple definitions of MAS [71]. According to Weyns et al. [119], MAS can be seen as a particular software architecture as described below:

Multiagent systems provide an approach to solve a software problem by decomposing the system into a number of autonomous entities embedded in an environment in order to achieve the functional and quality requirements of the system.

The above definition takes a top-down approach. It assumes there is a system that can be decomposed into interacting elements, the autonomous agents. It sees MAS as a specific decomposition of system elements intended to satisfy the system requirements. The definition focuses on the structure of the entities composing the system.

An alternative way to understand a MAS is to focus, instead, on the interactions between the agents participating on the system and their behaviour. Wooldridge [120] states that a MAS is composed of multiple and interacting computing elements, the agents. It states that these agents should have autonomous behaviour according to their design rules. Wooldridge definition also states that agents should also be capable of interacting with other agents. Agents collaborate to achieve the system goals.

Each definition provides a particular way to see a MAS. Each definition solves different needs of a system designer who wants to create a MAS, but do not interfere with the understanding of what a MAS is. On the one hand, using software architecture views can be very interesting for documenting a MAS. On the other hand, focusing on creating the interaction mechanisms between agents and understanding some possible emergent behaviour is also very useful for certain usages of a MAS.

Underlying both definitions of a MAS are the concepts of: i) autonomy, ii) interaction. Autonomy refers to the capabilities of the system to perform operations on behalf of its users, having some degree of independence.

Autonomous agents are software entities capable of autonomously performing tasks in complex and dynamic environments on a users behalf. They are situated in an environment, which can be a real environment, as in the case of autonomous robots, or a virtual environment, as in the case of planning travel agents. Nonetheless, the environment on which an autonomous agent is situated is crucial to designing a MAS system. An agent needs to be able to sense the environment and act upon it to achieve its goals.

A MAS is made by many agents, and not only a single one, thus interaction is a key aspect concerning such systems. Interaction is more than simply communicating with other agents. Interaction occurs when an agent's actions are perceived and reacted upon by another. The interaction between two or more agents can happen directly, via direct communication, using a network for example, or indirectly, via the environment, dropping information on a virtual black board.

2.3.1 BDI Agents

BDI agent's design are largely inspired by the principles illustrated by Bratman [15]. Bratman's research on human behaviour, centers around his model of how human beings organize themselves in a rational way. Bratman models human behaviour in terms of our Believes, Desires, and Intentions. Agents research borrowed the Believes, Desires and Intentions model from Bratman's work to architect BDI agents.

A BDI agent has a view of the world (its environment), which is based on its inputs and sensors, and has a number of ways to influence the environment via its actions. BDI agents have a feedback loop, which allows them to constantly monitor the environment and to adapt to changes on it. An agent's feedback loop is illustrated in Figure 2.3.

The view of the world constitutes the agent beliefs. It has a set of desires, which can be calculated by the agent given its design goals and current set of beliefs. The agent's intentions represent the current course of actions a BDI agent plans to perform to achieve its goals [82].

The high-level architecture of a BDI agent consists of a loop with functions to sense the environment, deliberate on a course of actions given the agent's goals, and executing such actions [82], [101]. It is very important to note that this continuous loop allows an agent to constantly receive feedback from the environment and to adapt its plans accordingly [82]. The continuous monitoring of the environment allows a BDI agent to create new plans when faced with sudden changes in the environment. This simple architecture allows a clear

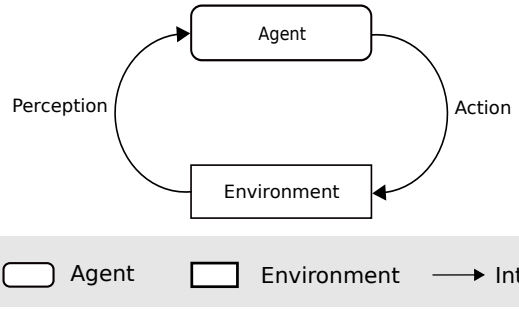


Figure 2.3: A BDI agent constantly monitors the environment, creates a set of beliefs regarding the current state of the environment, and plan accordingly to this current set of beliefs and to its own design goals.

separation of concerns between the different parts of an agent. That way, it facilitates to create specialized behaviours for information gathering, deliberation of plans, and to concretely execute the chosen actions. Algorithm 2 illustrates the main aspects of a BDI agent.

Algorithm 2 BDI agent

$B \leftarrow B_0$
$I \leftarrow I_0$
loop
  $\rho \leftarrow see - environment()$
  $B \leftarrow create - beliefs(B, \rho)$
  $D \leftarrow option - generator(B, I)$
  $I \leftarrow updates - intentions(B, D, I)$
  $execute()$
  $drop - successful(I)$
  $drop - impossible(I)$
end loop

BDI agents provide design principles for the internal behaviour of an autonomous agent. Autonomous BDI agents behaviour can express two extreme behaviours [58]. On one extreme, the autonomous agent can be cautious, meaning the agent constantly reconsiders its action plans and if there are better plans it will simply change its commitment and commit to the new plan. On the other extreme, there are the bold agents, which never deliberate again, once they decide on a course of actions. Both agent behaviours are useful given the problem at hand. Bold agents cope well with static environments, on the other hand, cautious agents cope better with dynamic environments [58]. Because cautious agents

constantly reconsider their options, it is possible that a cautious agent will find more suitable plans if its current plan is not optimal anymore.

None of both extreme behaviours, being cautious or bold, is very good for either static or dynamic scenarios. A cautious agent, can be trapped in constant reconsiderations if there are cyclic changes on the environment that can lead to plans constantly changing and no actual action been done. A bold agent can be stuck on a certain plan of action which is not feasible anymore, in the case the changes on the environment make the plan unfeasible.

Which type of autonomous agent is used on a system, heavily depends on the application domain and the level of autonomy designers can give to the agents. BDI agents constitute a very interesting design rationale for creating agents which have to cope with unexpected situations. A common critique to BDI agents is that is not easy to guarantee expected behaviours of an agent, however, there is research trying to tackle that [8], and it is always possible to limit the number of actions an agent can take at any moment.

The communication patterns between a group of agents is also a relevant aspect of a MAS design. In the following subsection we delve into agent coordination mechanisms, which deal on how to organize a group of agents in order to achieve a certain system goal.

2.3.2 Coordination Mechanisms

A crucial aspect of a MAS is the fact there are several agents operating on the system. Coordination mechanisms are used to ensure that the agents act in a coherent manner [73]. By coordinating agent's actions it is possible to prevent chaos in the system. Since no single agent has a global view of the system, it is possible that an agent actions will conflict with other agent's actions. Another reason to coordinate agent's actions is to meet global constraints, that is, constraints which should be met by the system as a whole.

The agents coordinate their actions according to certain patterns, designed by the designers of the MAS system [113]. There are different coordination patterns, normally called coordination mechanisms, that can be used in different systems. Since there are many ways to coordinate agent's behaviours, it is interesting to distinguish a few common coordination categories. A simple way to categorize coordination mechanisms is to distinguish them according to the constraints they impose on each agent behaviour.

Social Laws and Conventions

Social laws can be used to model an agent behaviour to avoid unwanted states on the system [45]. A typical example is when designing robots, one does not want that the robots collide with each other. As such, a social law can be to impose that an agent, in this case a robot agent, does not move to the same location already occupied by another agent. It is interesting to note that agents do not need to directly communicate to obey social laws. In the robot example, an agent obeying the social law of avoiding collisions, only needs to use its sensors to detect that there is another agent on a certain location and not move to that location. A benefit of using social laws is that it avoids unnecessary communication about system states that should never happen.

Following the same principle of encoding behaviour on the agents to avoid certain system states, it is possible to require certain actions to guarantee the system has certain desired states [90]. A **convention** tells an agent what it should do, given it perceives a certain situation in the system. For instance, a convention can be that every time a robot encounters another one coming in the opposite direction, it has to move to its right, in order to avoid a collision. Again, the agents would achieve a certain goal, moving freely by not colliding, by following a certain system convention. Another way to coordinate agents behaviour is via agent organizations [50] , which are explained next.

Organizational Structuring

Agent organizations are a way to structure agents behaviours in terms of the roles an agent has on the MAS. Organizational structuring borrows inspiration from human organizations, where there are a number of roles and people playing these roles [117]. Organizations are very effective at constraining the roles and the needed communication between the players of a role in order to guarantee the proper functioning of the organization. Imagine an organization where there are no roles for cleaning the buildings, for instance. It is clear, there would be an enormous coordination overhead between the workers to decide whom would be responsible for cleaning the building. An efficient human organization facilitates people to organize themselves according to the roles they are acting on the organization.

The idea of organizational structuring in MAS is that hierarchies can be created, optimizing the communication the agents have to perform to coordinate their actions. The concept of organizational structuring can be used, for instance, on a wireless sensor network. It is possible to distinguish roles as a `relayer`, or a `data aggregator` on a wide wireless sensor network. In such network, nodes

having more battery power could play the role of relayers, because using the radio demands a lot of energy from the node. Nodes having higher computing power could play the role of data aggregator, and so on.

Contracting

A third way to enable the coordination between a large number of agents is to create high level auctions which agents participate to achieve the goals of the MAS. Auction protocols are specially suitable to coordinate agent's actions on problems that can be decomposed in well-defined tasks, with very little coupling between these tasks [73], [111].

Contract-Net is a coordination mechanism based on the contracting approach. It is a market-based protocol that has being used in different applications [32]. Contract-Net is a standardized protocol used to allocate tasks to agents [93], allowing a distributed resource allocation strategy. Each participating agent can optimize its own resources while participating in the coordination.

The protocol was created inspired on the agency problem on a company. On a typical company, managers have to assign tasks to contractors (workers), whom perform the tasks. Alternatively, a manager can ask which worker is willing, or is capable, of performing a certain task and workers could reply stating when they could complete such task.

The Contract-Net protocol also resembles an English auction, in the sense that agents can announce tasks to all other agents and wait for bids. However, not like an English auction, the Contract-Net protocol stops after a single bidding round.

An agent, called the manager, announces a task to all other agents. The other agents, called contractors, see a task announcement and can then bid, only once, to perform the announced task. The manager agent waits for bids for a certain time period and assigns the task to the contractor agent with the best bid.

In principle the protocol is extremely simple, which is good, but still leaves some difficult decisions to the system's designer. The system designer has to decide how long a manager agent is going to wait for the execution of the protocol, for instance. This waiting time for bids is dependent on the particular system and can lead to completely different task allocations, for instance, if the waiting time is too short or too long.

The Contract-Net protocol has limitations on its scalability and issues related to the quality of the assignments, specially in dynamic systems. For instance, according to the protocol, an announcement has to reach all the agents on

the network, what can be costly on a very large network. Because of the above mentioned reasons, there are extensions to the Contract-Net protocol, as the DynCNET [114], which tackle Contract-Net short-comings in dynamic environments.

Nonetheless the Contract-Net protocols has been successfully applied on grid and cloud computing environments [94],[49], showing its efficacy for different coordination problems.

Local Planning Prior to Coordination

The main principle in local planning prior to coordination is that cooperating agents initially create their plans and later exchange information with other agents to check if each plan is feasible or not [56]. This technique can be seen as a type of "divide and conquer" approach to problem solving. Initially agents create their local plans, (divide) which is a simpler problem than creating a global plan which encompasses all agents. Then, the agents have to exchange information with each other to check the feasibility and possible conflicts of such local plans (conquer)

A problem of such approaches is that it is not possible to guarantee it is possible to find a optimal joint plan. The main steps used to implement this technique are:

- Each agent builds its local plan, disregarding the existence of other agents in the environment
- Agents interact with each other looking for possible conflicts in their plans
- Iteratively, each agent adds more constraints to its local plans, trying to avoid conflicts with other agents plans
- The coordination is done when the agents decide that there are no conflicts in their joint plan.

If there are still conflicts, some agents have to create an alternative local plan and start the checking for conflicts again.

A challenge in such approach is to decide how agents can identify and rectify problems in their joint plans. An approach, for instance, is to simulate the executions of each agent's plans and detect inconsistent states in the simulated environment.

Local planning prior to coordination techniques focus on fixing the joint plans wherever agents have finished their local plans. However, this process can be time consuming and not lead to a feasible resolution of conflicts.

2.4 Swarm Intelligence

Swarm intelligence is a computer science research field which uses the social insect metaphor for solving complex computational problems [12]. Biologists categorize a number of species of insects as social insects. These little insects posses limited individual capabilities to act alone, however a swarm of social insects presents a behaviour which is bigger than the sum of its parts. Social insects present very evolved behaviour, which is seen, for instance, in the construction of bee hives. Bee hive constructions is very organized and optimized to solve issues such as protection of the hive, temperature control, and food production.

Social insects behaviour inspires the creation of algorithms which are distributed, which rely on direct or indirect interactions, flexibility and robustness. Swarm intelligence deals with using social insects as a metaphor to create algorithms which present a collective intelligence which emerges from groups of very simple agents.

As systems become more complex and difficult to control, researchers need to find new ways to create systems which can deal with the complexity of very large scale, and complex systems. Example of such complex systems, are communications networks, robotics, and large combinatorial optimization problems [38].

2.4.1 Stigmergy

A common strategy of indirect communication found in nature is in the form of stigmergy. Social insects need to coordinate their actions in order to achieve their goals, be it a nest building activity, nest protection, or food foraging [102]. There are direct ways of communication like, antennation, mandibular contact, visual contact, chemical contact, etc. However, a more interesting form of communication, is done via the environment, when agents exchange information by modifying the environment in order to give information to others. This form of indirect communication via the environment is called stigmergy.

Argentine ants *Linepithema humile*, for instance, use stigmergy, besides other techniques, to indicate the quality of food sources. Whenever an ant finds a good source of food it backtracks to the nest dropping pheromones along its

path. Pheromones are chemical scents that stay in the air during a certain time, to indicate paths to sources of food. Other ants can "smell" such pheromones and follow the same path to the food source. As more ants follow a certain path, they reinforce the pheromone trail on that path, leading to even more ants to follow that particular path. That way, the ants "know" that a particular path is good or not.

However, not all ants follow the strongest pheromone trails. A certain number of ants "get lost" and continue exploring the environment searching better food sources. Ants mix exploratory and scent following behavior, in order to always have good sources of food. After a short time period, almost all ants in the nest know about the better food source and has a chance to reach it.

2.4.2 ACO - Ant Colony Optimization

ACO algorithms are algorithms inspired by food foraging behavior of real ant colonies. Ants are very good at finding the shortest paths between their nest and sources of food. They can even find the shortest paths between their nests and the best food sources in the environment [84]. However, individual ants have little capacity to find new paths individually, they collaborate with each other to solve their food foraging problem. Besides other clues, ants use stigmergy to indirectly communicate with other ants. They drop pheromones in the environment leaving pheromone trails where they walk.

In ACO algorithms a set of virtual ants, which are software agents, indirectly cooperate to find solutions to complex optimization problems [35]. The ACO meta-heuristic defines a number of steps to create algorithms that mimic ant behavior. It can be applied to any problem that can be reduced to path-traversal problems [36].

The virtual ants have well defined behavior in order to contribute to finding solutions. Virtual ants *drop* information, called pheromones, along the paths that lead to good solutions. That way, other virtual ants, can *smell* the pheromone trail and follow it as well, converging to a path that represents a good solution to the problem at hand. Pheromones in ACO are pieces of information about the quality of a particular path. In the traffic domain, a pheromone can model the quality of a particular route, for instance. Other cars can then use this pheromone information to decide if a route is crowded or not, or if the route is a good option to use or not.

More formally, ACO can be used with problems which can be represented a graph $G = (N, E)$ where N is the set of nodes and E the set of edges from the graph G. Even more, the solution to the problem can be represented as paths

on the graph G. Besides being represented as a graph, problems suitable to be solved using ACO are characterized by:

- A finite set of nodes $N = n_1, ..., n_n$.
- A finite set of edges $E = (n_i, n_j), ..., (n_k, n_l)$, where $\forall i, j, k, l \in Indexes(N)$.
- For each edge E, there is a cost function, possibly parametrized by time.
- A finite set of constraints is assigned over the elements of $NandE$.

The solution for a problem, solved using an ACO algorithm emerges from the indirect interactions of the virtual ants which look for the solution. An ACO algorithm is composed by a set actions that are taken by the virtual ants while finding a solution to the problem at hand. An ACO algorithm is made by creating a number of very simple agents, called the virtual ants, which have a clear behaviour. Virtual ants crawl the problem graph trying to find the best path according to their objective function. If an ant has to decide which new edge to follow, it decides which path to follow by: i) analyzing the pheromone level of a particular path, ii) analyzing the heuristic value for that path. The ant builds a probabilities table based on the pheromone information and the heuristic values and selects a path to follow based on this table.

When an ant finds a solution, it backtracks the path it has followed, and drops pheromones along this path. By dropping pheromones on a path, the probability that other ants will also explore solutions which incorporate such path is increased. After a number of iterations, all ants in the system will eventually converge to the best path to the problem at hand, leading to the desired solution.

The main shortcoming of ACO algorithms is that is is possible to create paths which lead to local optimums. To avoid being trapped in local optimums, it is possible to add other mechanism to the algorithm, as pheromone evaporation. Pheromone evaporation is the process of lowering the pheromone information levels recorded on the system, to avoid all ants converging too quickly to a particular path. That way, the ants will always have a chance to explore new paths which were not explored before.

Another shortcoming of ACO algorithms is that it is hard to give strong guarantees about the convergence of the algorithms, since it is a probabilistic approach to find solutions to problems.

2.5 Overlay Networks

Overlay networks are networks that are built on top of other networks [99]. A classical example of an overlay network was the beginning of the internet. The internet was an overlay on top of the telephony network. Nodes participating on the internet could communicate on ways not foreseen by typical telephony networks. Nowadays, telephony networks have changed to packet switching and, many times, operate on top of the internet network.

Clark et. al. gives a more precise definition of an overlay network [25].

"An overlay is a set of servers deployed across the internet that:

a) Provide infrastructure to one or more applications,

b) Take responsibility for the forwarding and handling of applications data in ways that are different from or in competition with what is part of the basic internet,

c) Can be operated in an organized and coherent way by third parties (which may include collections of end-users)"

The first item from Clark's definition states that an overlay network is more than a regular end-user application. An overlay network provides infrastructure to other applications. The second item, (b), touches the behaviour aspect of an overlay network. An overlay is responsible for routing application data in ways that are different than the underlying network. Finally, the above definition, makes clear that an overlay network is not managed by the same organizations which operate the underlying network. But, naturally, networks evolve with time and, an overlay may become part of the core network and be managed by core network providers, instead of third parties. Figure 2.4 depicts how the connections between nodes on the overlay are different than the connections between nodes on the underlay network.

It is possible to distinguish two basic types of overlay networks [17]. A distinction is made based on how the nodes participating on the overlay network are structured, which implicates on different overlay geometries. Unstructured overlay networks do not enforce any particular structure on how nodes are linked. Nodes participating in a `Structured Overlay` network have well defined rules concerning which other nodes they can be linked and, sometimes, which information new nodes will maintain.

`Unstructured Overlay` networks may resemble random graphs. On the other hand, `Structured Overlay` networks have well defined structures, like rings, hypercubes, etc. as is the case of Chord and CAN respectively [95, 83].

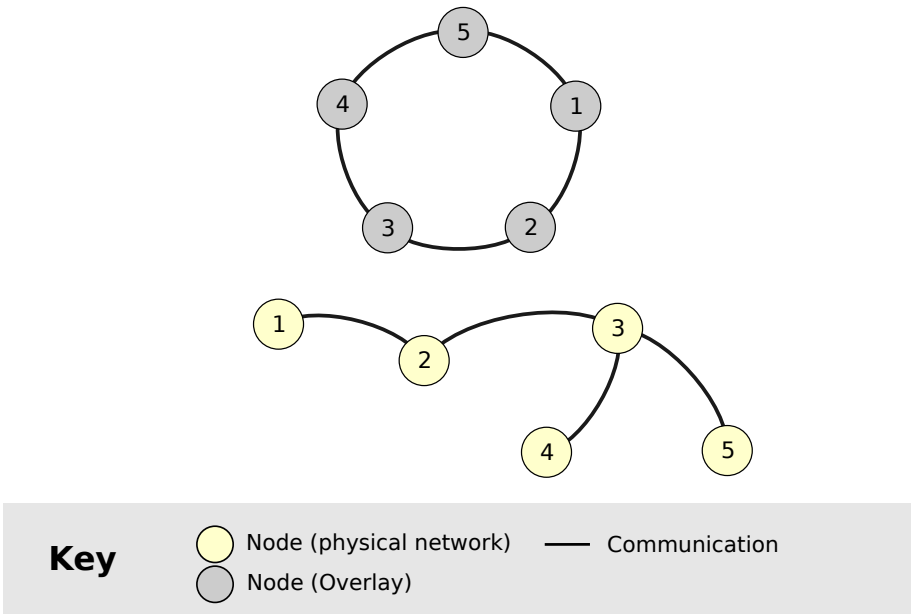


Figure 2.4: Nodes, on an overlay network, have direct neighbors that would, otherwise, be very far away on the underlying network. The yellow nodes represent the physical underlying network, while the gray nodes are connected on the form of a ring on the overlay network.

The different topologies result in different benefits for using such networks and on different maintenance costs. Performing queries for data stored on the overlay network is more expensive on unstructured networks, since the network does not maintain any information about optimizing queries. On the other hand, searching for information on a structured overlay is normally done within hop constraints, given the known structure of the network. A typical challenge of overlay networks is to maintain the overlay topology and to maintain an up-to-date view of a node's peers.

A way to guarantee, or increase, the probability to find a certain information on an unstructured overlay networks, or to help maintaining the overlay topology is to use gossip protocols [57]. Another very common usage is to use gossip protocols to gather quality information about nodes on an overlay network. Gossip protocols, also called epidemic algorithms, are algorithms that help

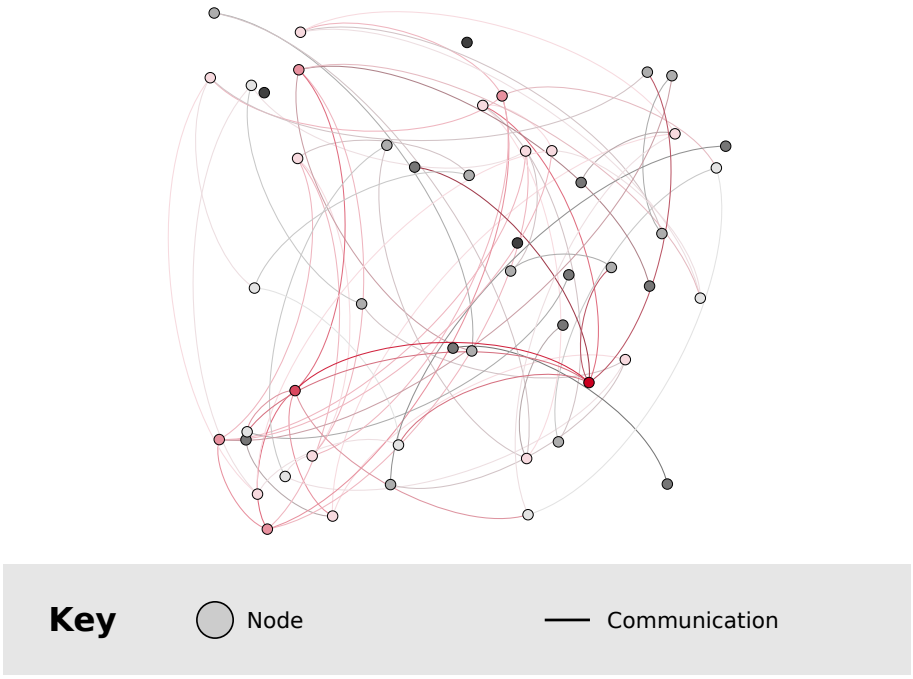


Figure 2.5: An unstructured overlay may resemble a random graph. Certain nodes can have a high degree of connections, while other may have only one connection, what can make querying the overlay more resource consuming.

spreading information between peers on a network. The main characteristic of gossip protocols is their probabilistic nature and simplicity. Basically, a gossip protocol consists of continually randomly selecting peers, exchanging a piece of information with them and repeating the processes while the system is operating.

Different gossip protocols, like Cyclon, Clon, T-Man, have different characteristics in terms of communication overhead, response time, resilience to churn, etc [68, 55]. Cyclon is a very simple gossip protocol that we use to exemplify typical epidemic protocols [110]. Each peer knows a subset of other peers, its *neighbors*, which are constantly changing. From time to time a peer shuffles its list of neighbors, by exchanging its list of neighbors with a randomly selected neighbor. The neighbors list shuffling happens often in every peer. A illustrative pseudo-algorithm for the peer shuffling is depicted below:

1) Select random subset S of neighbors.

2) Select a random peer p from the selected subset of neighbors S.

3) Send S to p.

4) Receive a subset R of peers from p.

5) Discard duplicate entries from R and elements to its own list of neighbors.

When a peer receives a request to exchange peers, it acts as follows:

1) Receive subset S of peers.

2) Select random subset R of neighbors and send it to requesting peer.

The different gossip algorithms also lead to different information propagation speeds. In our research, we are interested in guaranteeing fast propagation of quality information on the network to avoid stale quality information on the peers.

2.6 Conclusion

In this chapter we presented the main concepts needed to understand our research. We briefly described web-services concepts, both WS-* and `RESTFul` services. We also explained the concepts behind service composition in the form of composite services. Our research borrows concepts from the MAS field, which provide several techniques and concepts to create autonomous systems made by hundreds of entities. We touched the different forms of coordination in MAS. We briefly introduced the concept of Swarm Intelligence and ACO. To conclude the chapter we explain the notion of overlay networks and gossiping protocols, which are also used in our research.

Chapter 3

Assumptions and System Model

In this chapter we first delineate the system, then we highlight our assumptions regarding the environment where the system operates. Because part of our research focuses on creating robust systems, we are also interested in the system behaviour under failure, specially massive failures. To that end, we dedicate a section to explain the failure models we have used.

Our assumptions can be divided in three main areas: failures, communication, and information consistency. Each section explains each of these assumptions and its consequences on the system.

Our assumptions constrain possible infrastructures and henceforth applications that can be created using our techniques. This chapter's goal, is to make it clear which are these assumptions about the system model and its underlying infrastructure.

3.1 What is "the system"?

We consider the system to be constituted by a number of `Services` and `Service Managers`. `Service Managers` are responsible for bookkeeping the usage of their `Services` and interacting with other `Service Managers`. `Services` provide the operations needed by other `Services` in the system.

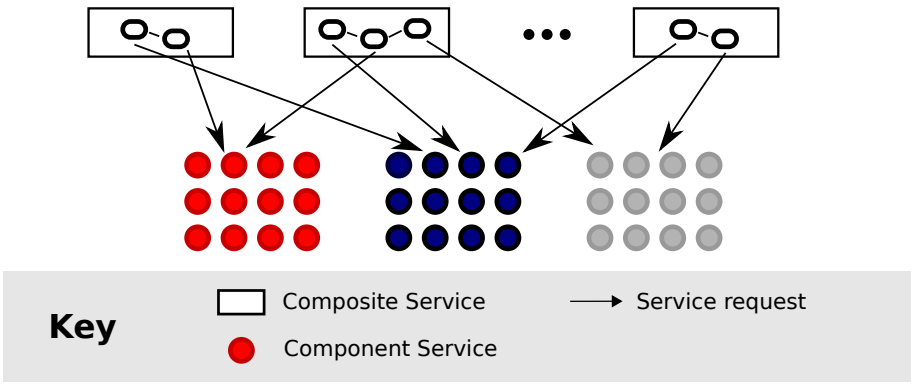


Figure 3.1: We assume there are a large number of component services providing the same operations. Component Services, or simply services, providing the same operations are said to be of the same type T, and to be *replica services*. The figure depicts three sets of *replica services*, each set having a different color, and a number of composite services using different replica services.

The system is open to new services becoming available and services shutting down at any moment.

A `Service` and its `Service Manager` are software entities residing on the same computing node. A `Service` provides the operations that are invoked via the network, by other services. For instance, an operation can be to perform an image transformation, or to execute an algorithm. A `Service Manager`, on the other hand, is responsible for maintaining the information about the availability, and the quality of the services which they are associated with. We assume that there are multiple services, also called *replicas*, providing the same operations. A *replica* service has varying qualities, such as a different response time, or cost, which can be evaluated at runtime.

`Composite Services` work by mainly composing the operations of other services. For now, we only have to keep in mind that a number of services in our system belong to the category of `Composite Services` and they are created by composing the operations of other services. Figure 3.1 shows an abstract view of three sets of *replica* services, being used by a number of `Composite Services`.

A `Service` has a type T, called its *abstract service type*, which is defined by the set of operations it provides. Several `Services` can have the same *abstract service type* T, and the number of `Services` or the *abstract service*

types available on the system is not known a-priori.

More formally, we define the system Σ as $\Sigma = (S, \delta, T, M, \mu)$, where S is the set of n `Service` instances $\{S_1, S_2, \ldots, S_n\}$. The mapping $\delta : S \to T$ maps each `Service` instance to a particular abstract service type T, where $T = \{T_1, T_2, \ldots, T_m\}$, is the set of abstract service types. M is the set of n `Service Manager` instances $\{M_1, M_2, \ldots, M_n\}$ and the bijection $\mu : M \to S$ associates one `Service Manager` to one `Service`.

We assume the services to be cooperative in the sense that a service does not need to protect itself from malicious services.

The system operates in a dynamic environment. The system is open to services entering and leaving it at any time and the number of execution requests also change constantly. It is not possible to know in advance how many services will be interacting in the system, at any given time, or how many service execution requests there will be. Another source of dynamism is that several problems can happen at any moment, services may fail, the network can become unresponsive or even partitioned, or service execution requests can arrive in pikes.

3.1.1 Services and Service Managers

A `Service` is a software entity that has a well defined interface, and offers operations which are invoked over a network. Services can be created in any programming language, as long as they provide proper interfaces to be invoked via the network. Each `Service` has a corresponding `Service Manager`.

`Service Managers` maintain quality information about the services they are associated with. They maintain information about the execution schedule of their services. Every time there is an execution request for a service, the `Service Manager` can add this request to a list of intended requests and can calculate the probable amount of time it will take to execute such request. The concrete invocation of a service is made directly from a service to another, not necessarily passing by the `Service Manager`. Once a request execution is finished by its `Service`, the `Service Manager` collects information about its associated service performance. Possessing the information about a service performance and its schedule, a service manager is capable of answering future service requests with an increasing accuracy.

Service managers can directly communicate with their services and with other service managers, forming what we call a *coordination layer*. Figure 3.2 depicts four `Services` and their corresponding `Service Managers` on a computational environment.

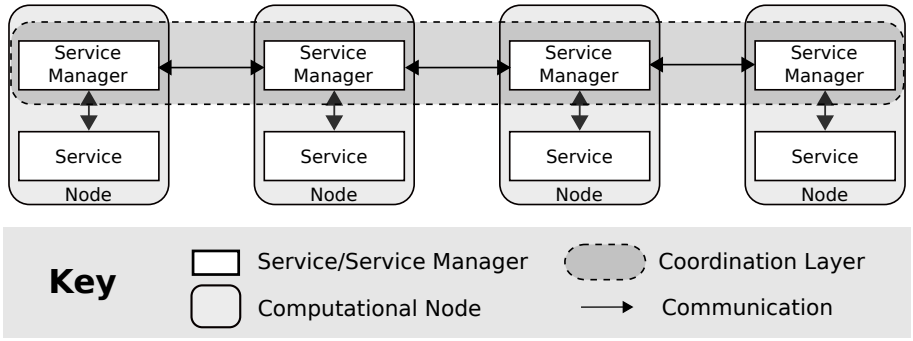


Figure 3.2: Services and Service Managers sit on the same computational node. Service Managers create an overlay network containing quality information, indicating good and bad services alike.

Service managers query the other service managers looking for services of a particular type, having a desired quality. The quality information about all the services is maintained by the Service Managers at the `coordination layer`. Service managers are not necessarily reactive entities, though. They can proactively propagate quality information in the *coordination layer* about their services as well.

Services on the other hand only provide operations to other services, when requested to do so. Unless the service is a composite service. A composite service completely relies on other services operations, not working if there are no available services providing the operations its needs.

3.2 Failures

An important aspect of our work is creating robust service compositions. As said before, services operate in an intrinsically dynamic environment, nodes can become inaccessible, fail, etc. An open system operating with a large number of services has to be able to cope with constant failures of a number of services participating in the system.

We assume nodes can fail, by crashing or losing their network connection, and later may resume to their normal operation, which is also known as the *crash-recovery failure model* [2], [92]. We consider that nodes do not arbitrarily misbehave, that is, they do not generate wrong data or send faulty messages to other nodes, such as what can happen in Byzantine systems [37].

We do not model partial node failures, such as a service failing but its service manager staying alive. In our model, we assume that when a node fails, the service and service manager operating at that node also fail. We also assume that composite services only fail if they can not find all needed component services they need, within a certain time.

We are not interested in individual node failures, but, instead, in the effects that several failures may have on the system as a whole. Hence, we focus on the effects on the system of failures that affect several nodes at the same time. We say that when several nodes are failing at the same time, the system is being disrupted. We define the disruption of the system according to the percentage of nodes in the system which are failing in a given moment.

- Small disruption. Less than 20% of the nodes of the system are broken.
- Large disruption. More than 80% of the nodes of the system are broken.

The first type of disruption only affects a small percentage of nodes, and may have a negligible impact in the composite services which rely on the broken services, which were running in the broken node. The second type of disruption affects a large number of nodes at the same time, possibly breaking the entire system.

We model the failures in our system by stochastically removing nodes from the system using different probability distributions. We model node failures using the **Uniform** probability distribution. We assume that all services fail according to the same probability distribution. The duration of a failure is modelled using a **Poisson** distribution, having a very small probability of a service not recovering from a crash.

Failures can be seen as a function $\epsilon : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ which takes a set O of services, where $O = \{S_\alpha, S_\beta, \ldots, S_\gamma\}$, $O \in \mathcal{P}(S)$ and returns another set $O' \in \mathcal{P}(S)$. The function ϵ randomly removes, using the **Uniform** distribution, elements from the input set O with the probability ρ, which can be either 0.2 or 0.8. Each node fails for a certain time and resumes its operations. The individual node duration is given by the **Poisson** distribution with mean δ, which is defined for each experiment.

3.3 Communication

The system relies on a communication network providing communication between the different services operating on it. Thus, we model the

communication of our system accordingly. We assume that there is an underlying network providing communication capabilities to the services and service managers from the system. The communication between the entities of the system is done via sending messages in any directions between two nodes on the network. We assume the network to be unreliable, that is, messages sent over the network can be delayed, lost, duplicated, or arrive out-of-order at the desired recipient. We do not rely on any multicast mechanisms from the underlying network, we only assume it is possible to send messages to particular recipients, given they have an address, i.e. an `IP` address.

3.3.1 Assumptions regarding the order of events

Distributed systems models are sometimes divided in two categories, according to the assumptions regarding the communication characteristics of the system : `synchronous` and `asynchronous` [60, 98]. However, we do not believe there is such binary dichotomy on real life distributed systems. On the contrary, we believe that there are many variants. For instance, there is the timed asynchronous distributed system model [28], and our system model. The `synchronous` and `asynchronous` models help to reason about the expected behaviour of a distributed system model in question, though.

The synchronous model assumes that there is a known upper bound delay for message delivery. It also assumes it is possible to define lower and upper bound limits for the execution of processes on the system. A process is any kind of computation that takes place on a node of the distributed system and that produces results based solely on its current state and the information contained on the message it has received. The synchronous model also assumes that the drift between local clocks has a known bound.

The asynchronous model, on the other hand, has less restrictive assumptions than the synchronous model. The asynchronous model does not assume limits for the execution time of a process. There are also no assumptions on the time it takes for a message to be delivered and no assumptions on upper limits for the time drift between local clocks of each process.

Our model, as the asynchronous model, does not assume upper limits for the execution time of processes, or known bounds on the delivery times of messages. However we do assume that each process, an agent in our terminology, has access to a local clock. We assume the possible time drift between the processes to be orders of magnitude smaller than the needed time resolution for our system to operate properly. Even more, we assume the each service has access to a time server, as NTP for example, limiting the total time drift between

Property	Assumption
execution time for the processes	no lower or upper limits for the duration of a process execution
access to accurate local clock	each node on the system has a local clock, which is precise and synchronized with other clocks via NTP

Table 3.1: Our assumptions regarding the communication in our system model

local clocks of the nodes where the services are executed. We summarize our assumptions regarding the communication in our system model, in Table 3.1.

3.3.2 Consistency

Another consequence of having an unreliable underlying network infrastructure is that besides nodes failing, communication links can fail, which can lead to inconsistent state in the remaining nodes.

Depending on the particular topology of the communication network, it is possible to have network partitions due to a failure of a certain number of links. Network partitions lead to inconsistency on the system, since different nodes can posses divergent views regarding the same information.

The system model we use copes with the consistency problem at local levels. Service managers keep quality information about a subset of other service managers. However there is no global state that has to be consistent across all different nodes in the system.

We assume network partitioning can happen, though we do not expect it to be common, due to the nature of internet scale environments. The system provides eventual consistency on its coordination layer [109]. This means that different service managers containing quality information regarding the same service will eventually return the same information, if there are no updates to the quality information. However we do not make strong assumptions for the time it takes for the system to stay on a consistent state again.

3.4 Conclusion

This chapter highlighted the main assumptions regarding our system. We show our assumptions regarding how the services fail and recover. We also explain our assumptions regarding the expected communication between the services and service managers.

It is worth mentioning that securing distributed systems is a broad research area. Security research varies from focusing on the network, on the application architecture and development, to runtime approaches [121], [16]. The security aspects concerning the system are outside the scope of our research, since securing distributed systems consists of an important but different research subject.

System designers have to keep in mind the assumptions exposed in this chapter, especially the eventual consistency from the coordination layer, if they decide to use our techniques. The next chapters explain in details our approach to create robust composite services in a decentralized fashion.

Chapter 4

Approach

Coordination and control systems represent a family of systems whereby a decentralized software controls the functioning of the underlying, virtual or physical, system. Possible examples applications are manufacturing control [115], traffic congestion avoidance systems [21], and collaborative charging of electrical vehicles [106].

Coordination and control systems have similar characteristics such as:

- There is a large number of interacting entities in the system.
- There is no single controller which can dictate the activities of all other entities in the system.
- The system has a very dynamic underlying operating environment.
- There can be multiple organizations interacting in the system.

Coordination and control systems are task oriented [86]. Entities have to perform tasks using the resources available in the environment. Typically, the software of a Coordination and control system operates at orders of magnitude faster than the entities of the underlying system. The entities of the underlying system are situated in an environment, which in itself is very dynamic. For instance, in an underlying system as a cloud datacenter, resources can break, new resources can become available, network links may fail, etc.

Service applications, for instance, are constituted by a large number of services, which in turn, belong to independent providers and organizations [39]. Each

organization controls its own services and do not accept to give up control to a third party to decide when a web-service should accept or not a new request. Service applications form a complex system, which operates in a large distributed environment. As the other domains studied in our research group, service applications do not support a centralized solution. Service applications require the coordination of activities in order to achieve the system wide goals.

The research team where this research was conducted studies mechanisms for Coordination and Control Systems in various domains. The research group developed a high level coordination mechanism called Delegate MAS which provides the basic abstractions that we have used and extended in our research.

4.1 Delegate MAS

Delegate MAS is a MAS mechanism designed to work with "Coordination and control systems" [118]. It allows the creation of systems which are decentralized, meaning that there is no single entity responsible for all the decisions in the system. The Delegate MAS mechanism relies on autonomous entities, the agents, which sense the environment and take decisions on which actions should be performed in order to achieve their design goals. The agents do not take actions individually, they coordinate their actions with other agents, in order to avoid conflicts for instance.

Delegate MAS relies on four main concepts:

- Decentralization.
- Autonomous entities.
- Coordination of actions.
- Indirect communication.

Decentralization

Differently than simply distributing a system between different computers, a decentralized system has separate entities which work with local information and communicate with other entities in the system in order for the system to work. Entities in a Delegate MAS system only operate with the information they have locally. This information is brought by the indirect exchange of information between the agents and by sensors in the environment. Since entities only have

a partial view of the information available in the system, and this information may be replicated in different entities, if a single entity stops working, the system may still continue working.

Autonomous Entities

There are two basic abstractions used in Delegate MAS. Abstractions which model an entity that needs to perform a task, and entities providing the resources needed for the execution of tasks. An entity which provides resources in the system is modelled as a `ResourceAgent`. A `ResourceAgent` represents the resources available in the system and is responsible for informing the system about the current, and future, state of its resources. In a traffic system, for instance, a `ResourceAgent` can model a road which provides space for cars to travel on it.

The second basic abstraction in Delegate MAS is the `TaskAgent`. A `TaskAgent` represents an entity that needs to perform a task in the system. A `TaskAgent` does not have all the needed resources to execute its task, it relies on resources provided by other entities. Thus, in Delegate MAS, a `TaskAgent` relies on using the resources provided by `ResourceAgents`. For instance, in a traffic system, a car can be represented as a `TaskAgent`. The task of a `TaskAgent` in the traffic system can be to drive from one location to another, minimizing the travelling time. If one wants to model this traffic system using Delegate MAS, one could model the many roads as a network of `ResourceAgents`, and could model the many cars as `TaskAgents`.

Coordination

The main idea of having `TaskAgents` and `ResourceAgents` is to provide a clear separation of concerns. Each agent type has a well defined behaviour in the system. `TaskAgents` and `ResourceAgents` coordinate together their actions to, for instance, better use the system resources. The coordination of `ResourceAgents` and `TaskAgents` actions follow the principles of intention propagation, sampling, and decay or reinforcement of information.

The intention propagation principle is based on the idea that a way to create robust plans in a distributed environment, is to share one's intentions with others [53]. That way, others can adjust their plans accordingly, taking one's intentions into account. For instance, if I plan to write a chapter of my PhD thesis at a particular day, I am better off sharing this intention to my close family and friends. That way, my close family and friends will know they

shouldn't try to arrange a party with me on that particular day. Because of this intention sharing we could agree on plans that are less prone to fail.

In a Delegate MAS system, `TaskAgents` share their intentions in the environment, so that other agents, such as the `ResourceAgents` can better optimize their own resources, and other `TaskAgents` can improve their own plans. The intention sharing is never done directly between one `TaskAgent` and another. `TaskAgents` always share their intentions in the environment, via the `ResourceAgents` for instance, so that any other `TaskAgent` can consult and react to such intentions.

A Delegate MAS system is designed to work in very dynamic environments. Thus, it never assumes that it is possible to have a static view of the system resources and environment that is always accurate. In order to deal with a dynamic environment, Delegate MAS relies on constantly sampling the environment. Agents sample the environment to create a view of the available resources and their state.

A `TaskAgent` does not assume that it can simply use a particular `ResourceAgent`, or even more, that a particular `ResourceAgent` exists. A `TaskAgent` samples the environment, searching for available `ResourceAgents` to execute a particular task. As the environment may change, the plans created by a `TaskAgent` may also change to reflect those changes. That way, the chances of a `TaskAgent` achieving its design goals are greater, than in a system where entities assume complete and fixed knowledge of the environment where they operate.

Information decay and reinforcement helps to lower the amount of stale information in the system. A `TaskAgent` can indirectly inform other `TaskAgents` about the quality of particular `ResourceAgents` in the system. This quality information may be accurate for a particular moment in time, but may be inaccurate after a certain time has passed. A way to deal with such stale information is to let information decay, that is, have less value as time passes. That way, other agents in the system, have more information about the quality of the information they are using.

The reinforcement of information occurs as a way for `TaskAgents` to indicate their willingness to use a particular `ResourceAgent`. In order to avoid having blocking reservations, as in leases in other protocols, a `TaskAgent` indicates to `ResourceAgents` how much it is willing to use a particular agent. This intention to use a `ResourceAgent` decays with time, so `TaskAgents` have to reinforce the intention information if they really want to use a particular resource provided by a `ResourceAgent`. If a `TaskAgent` decides it is not willing to use a `ResourceAgent` anymore, it does not have to inform the involved `ResourceAgent` since the information at the `ResourceAgent` will simply decay

and other agents will have the opportunity to use the resource provided by that `ResourceAgent`.

In summary, the benefit of having information decay and reinforcement is that it diminishes the amount of stale information existing in the system at any time.

Indirect Communication

Agents in a Delegate MAS solution never communicate directly with each other. Instead, the agents communicate via "smart messages", called ants in Delegate MAS terminology.

Smart messages posses the behaviour to explore the environment, to propagate intentions, and to check for feasible usages of resources. A smart message is a software entity which hops from one resource agent to another and can query the resource agent and its local environment. The smart message uses the queried information to decide to which other `ResourceAgent` to go to, once it has queried a `ResourceAgent`.

4.2 DMAS in Service Selection and Composition

We use ACO concepts to enhance our coordination mechanism regarding its resilience to failures and to lower the amount of communication needed by the agents in the system. The pheromone concept, for instance, is used to avoid direct communication between several agents, and to indicate the past quality of a certain set of resources. Thus, some abstractions as the `ExplorationAnts` and `IntentionAnts` mimic the ant colony foraging behaviour seen in nature and in ACO algorithms.

Our approach relies on two main abstractions which represent composite and component services in the system.

Component services, which mainly offer operations to other services, are represented by `ResourceAgents`. Composite services, on the other hand, mainly delegate their operations to other services, which provide the real resources to execute the needed operations. Figure 4.1 highlights the specialization of a service as a component or composite service, and the specialization of an agent into resource or task agent.

Composite services are made of a number of dependent activities. Each activity describes a particular operation that has to be fulfilled by a component service. The activities of a composite service form a graph, where each node represents

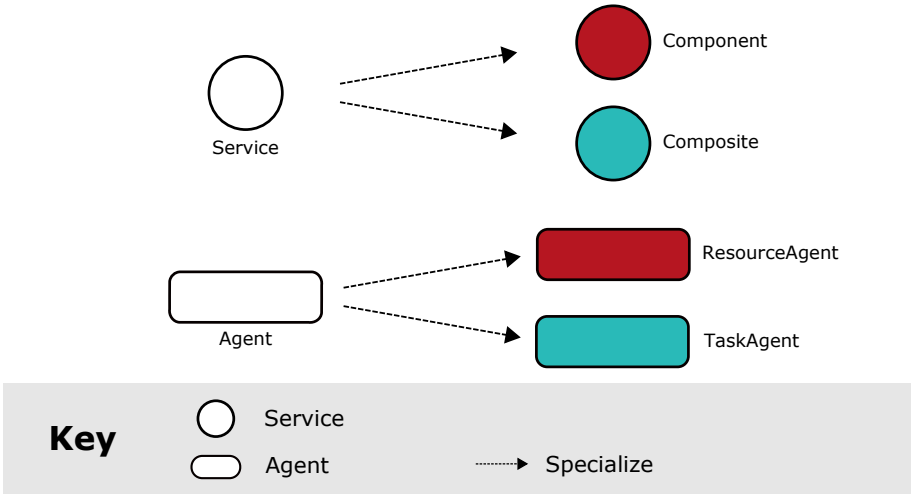


Figure 4.1: We model services as component, which have primitive operations, and composite, which require operations from other services. The agent abstraction is also specialized into two types of agents, either the Resource or the Task agents.

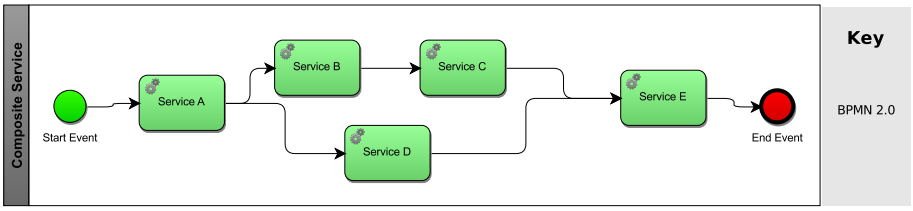


Figure 4.2: A composite service is described as a graph of activities, where an activity has to be fulfilled by a service. The graph structure also implies an order of execution of each activity, possibly having serial and parallel activity executions.

an activity and the edges represent the data-flow between the activities. An activity can only be fulfilled by services providing operations compatible with the activity. Henceforth this graph implies a particular order of execution of each activity, describing serial or parallel executions, as illustrated in Figure 4.2

Composite services are represented in our approach by the `TaskAgents`. For each service, either a component or composite, in the system there is one `ResourceAgent` or one `TaskAgent`. Figure 4.3 illustrates the main

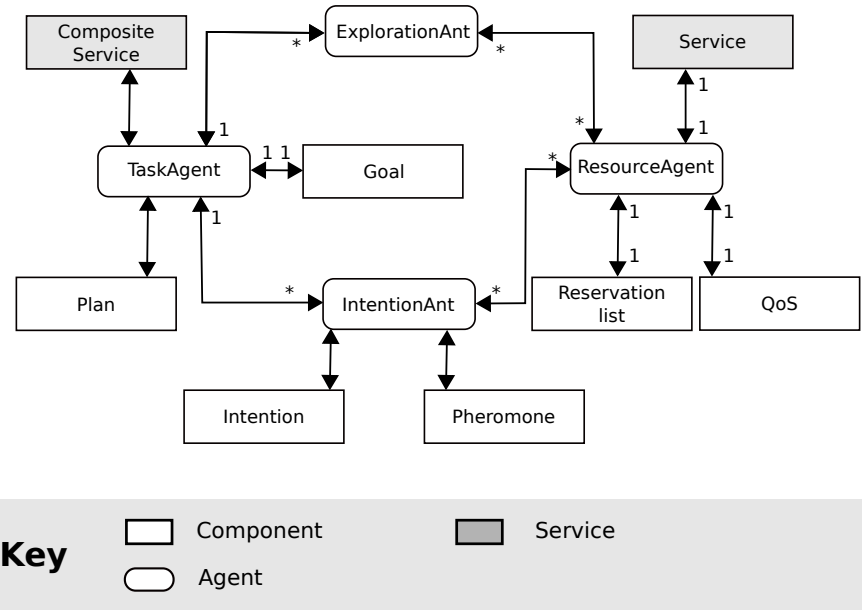


Figure 4.3: This figure shows the main abstractions used in our approach. `TaskAgents` are responsible for the proper invocation flow of component services from a composite service. `ResourceAgents` represent component services. The information about the quality of a particular component service is spread via pheromones in the system.

components used in our approach, highlighting the connections of `TaskAgents` with composite services and `ResourceAgents` with component services. It is also possible to see that the communication between `TaskAgents` and `ResourceAgents` is done via `ExplorationAnts` and `IntentionAnts`.

There is no central registry of services in our approach, thus, initially, composite services in the system are not aware of how to find the component services they may need. Instead of assuming the existence of a central registry that could contain information about the services interacting in the system and their qualities, our approach relies on a set of agents. Each agent is responsible for a tiny amount of information in the system. That way, if a particular agent stops working, the system can easily continue working without it. For instance, `ResourceAgents` are responsible for keeping track of the usage of their resources (services) and the quality of such resources.

By assigning responsibilities to different entities in the system, and creating ways to quickly find which entity is responsible for which information, we avoid having single point of failures in the system. In our approach, we also assume that all the agents participating on the system are cooperative, what means that no agent will try to cheat the system to receive more requests if it evaluates it can not really service the proposed request.

In the following sections we explain each concept involved in our approach.

4.2.1 TaskAgents for Composite Services

`TaskAgents` represent the composite services in the system. The main responsibility of a `TaskAgent` is to provide smooth operations to its composite service. A `TaskAgent` is responsible for:

- keeping track of the execution of a composition.
- checking which component services are needed for its composite service.
- selecting which component services are suitable to use.
- guaranteeing that a selection respects the desired qualities specified by the composite service.
- finding new component services to participate in the composition, in the event of failures of a component service involved in the composition.

A `TaskAgent` is a BDI agent. It creates its beliefs regarding the environment by constantly sending out `ExplorationAnts` which bring back information about the services environment. Then, a `TaskAgent` decides on which course of actions to take and creates an `IntentionAnt` which will engage on the plan.

A `TaskAgent` constantly inspects the network, looking for `ResourceAgents` that may be better alternatives for the future service invocations. It looks for component services to use, by sending out a number of `ExplorationAnts`.

The rate at which a `TaskAgent` sends out `ExplorationAnts` is determined at the system wide level and affects two different aspects of a Delegate MAS system. The first affected aspect is regarding the stability of the system. If a `TaskAgent` re-evaluates its commitments at a high frequency, there is a probability that the system will become unstable. In the above mentioned case, `TaskAgents` could keep alternating between a set of `ResourceAgents` in a cyclic way. Sending `ExplorationAnts` at a very high frequency may lead

to increased communication between the agents in the system, which, in an extreme situation, can saturate the network and make the system unresponsive.

Because of the above considerations, the rate of sending out `ExplorationAnts` is part of the commitment strategy used by the `TaskAgent`, and has to be fine tuned for each particular network. For instance, a strategy is that a bold committed `TaskAgent` sends less `ExplorationAnts` than a weakly committed `TaskAgent`, since it does not constantly re-evaluates its current intentions.

`TaskAgents` receive back, from `ExplorationAnts`, plans containing `ResourceAgents` addresses, that the `TaskAgents` can use for each activities of their composite services. The `TaskAgent` is responsible to select one of such plans. After selecting a plan, the `TaskAgent` informs the involved `ResourceAgents` and sends the addresses of the involved component services to the composite service. The `TaskAgent` delegates the task to inform the `ResourceAgents` involved in its plans to agents called `IntentionAnts`. `TaskAgents` have the following main properties:

- receive composite service description as input
- receive a SLA contract specifying the desired qualities for its composite service
- follow the execution of an instance of a service composition
- select possible component services to provide operations to its service composition instance, creating what we call a plan of execution
- inform the `ResourceAgent` from the selected component services that it intends to use their services at a particular time
- keep inspecting the network for possible alternative component services, while its service composition is being executed
- evaluates if there are better component services to participate in the composition, and decide to use them or not, what is determined by the `TaskAgent` commitment strategy
- upon the completion of is service composition instance, the `TaskAgent` spreads information about the quality of the engaged component services

As mentioned before, a `TaskAgent` is created following a BDI architecture. The `TaskAgent` operates in a asynchronous way, and is modelled using a finite state machine, as illustrated in Figure 4.4.

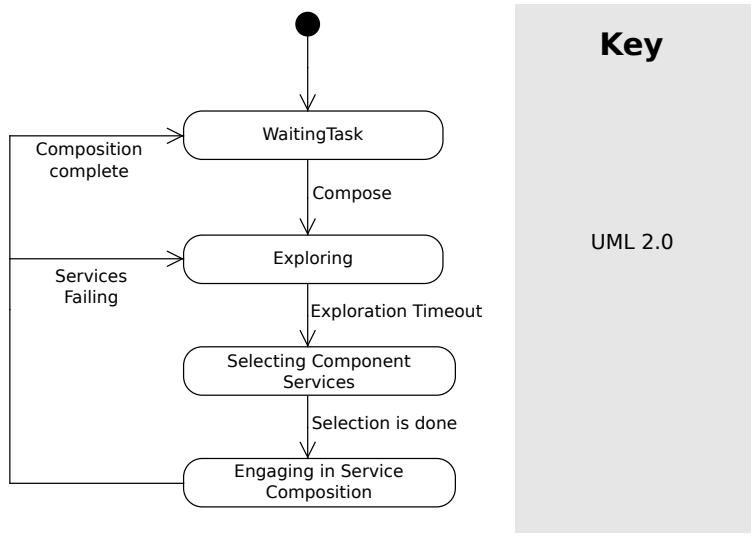


Figure 4.4: Initially a `TaskAgent` waits for a trigger to start looking for component services in the network. Since it delegates work to `ExplorationAnts` the Exploration state is finished, either, after a certain timeout or when all `ExplorationAnts` return a result. When a service composition is completely executed, the `TaskAgent` returns to a waiting state, being ready to start a new service composition.

A `TaskAgent` creates its beliefs about the environment during the Exploration state. At this state the `TaskAgent` learns about the QoS of component services in the system. During the Selecting Component Services state, the `TaskAgent` creates its intentions, which are specified in the form of a plan. This plan has the needed steps the `TaskAgent` has to take in order to fulfil its objectives. More specifically the plan has the same graph structure of the composite service definition, as illustrated in Figure 4.2. The plan associates a component service to a task and specifies at which time each component service should be invoked.

The `TaskAgent` creates a plan by selecting the best plan from a number of plans created during the Exploration state. The selection of the best plan is done by evaluating the goal function on each plan. The goal function evaluates the QoS parameters according to the relevance a system designer gives to each parameter. The QoS is represented by the vector $q = (q_1, \ldots, q_n)$, where $q_i \in \mathbb{R}$ represents the quality i. The quality can be, for instance, the duration time to execute one operation, price, trustiness, etc. The fairness is a quality attribute that can not be directly modelled using the quality vector. Fairness is achieved by the design of the approach, specially with the usage of the pheromones, pheromone

evaporation, and with the probabilistic behaviour of the exploration ants.

The relevance a designer gives to each quality parameter is represented by the vector $\lambda = (\lambda_1, \ldots, \lambda_n)$, where $\lambda_i \in \mathbb{R}$ represents the weight given to the quality i.

The objective function η we use for evaluating the QoS of services is defined below, (note that the smaller the λ the most relevant the quality is):

$$\eta : (q_1, \ldots, q_n) \times (\lambda_1, \ldots, \lambda_n) \rightarrow \mathbb{R}, \text{ such that}$$

$$\eta((q_1, \ldots, q_n), (\lambda_1, \ldots, \lambda_n)) = \frac{1}{\sum_i^n \lambda_i * q_i^*}, q_i^* \text{ is the normalized } q_i$$

When the `TaskAgent` is executing its selected plan, in the Engaging Service Composition state, it constantly monitors the remaining services in the plan. If it detects anomalies, such as a failing service, it can reevaluate the current plans and decide to switch to the Exploring state, to find alternative services. This behaviour is constrained by the commitment strategy used at the `TaskAgent`, which is further explained in the Section 4.2.6.

4.2.2 Resource Agents

`ResourceAgents` represent component services in the system, each `ResourceAgent` has an associated component service. The main responsibility of a `ResourceAgent` is to answer "what-if" questions. A "what-if" question is used to query a `ResourceAgent` about what would happen if another agent decided to use a resource at a certain time.

`ResourceAgents` answer "what-if" questions using the information they have about the past and future executions of the component services they are associated with. Since a `ResourceAgent` works very closely with its component service, the `ResourceAgent` has information about the past executions of its component service allowing it to estimate the average duration of past service executions.

A `ResourceAgent` also receives intention information from other agents. An intention is a piece of information stating that an agent intends to use a `ResourceAgent`'s component service at a particular time in the future. `ResourceAgents` use the intentions and the average duration of previous service executions information to answer the "what-if" questions the `ResourceAgent` receives. The `ResourceAgent` has enough information to create accurate short

term forecasts about the future load of its component service, what makes the answer to a "what-if" question more valuable to the asking agent.

`ResourceAgents` help to create the distributed infra-structure needed to coordinate the actions of the various `TaskAgents` in the system, by having pointers to other `ResourceAgents` in the system. That way, `TaskAgents` can explore the network of `ResourceAgents`, querying `ResourceAgents` and finding new ones, avoiding the need for a central registry.

A `ResourceAgent` has these main responsibilities towards the system:

- answer "what-if" questions.
- create short-term forecasts of their associated component service load.
- maintain the information about the QoS of their associated component service.
- provide pheromone storage for pheromones dropped by other agents from the system.
- book-keep a reservation list, which has information about which other agents want to use its resources (component service) and remove any stale information about other agent's intentions.
- maintain a table of neighboring `ResourceAgents`..

The behaviour of a `ResourceAgent` is summarized in the pseudo-code below:

The pseudo-code illustrated in Algorithm 3 shows that a `ResourceAgent` continuously receives requests from other agents and react to them. The `ResourceAgents` are also responsible for maintaining the information about pheromone levels dropped at them. The pheromone level indicates how often a certain `ResourceAgent` is used and can help other agents to evaluate if they should or should not use a certain component service.

At a technical level, a component service needs to provide an inspection interface that a `ResourceAgent` can use to retrieve information about which other services are currently engaging with the component service and when exactly this engagement started. Having this information, a `ResourceAgent` can continuously improve its view upon its associated component service, creating then better forecasts for service execution duration, and the time a `TaskAgent`'s service would need to wait before engaging with its component service.

Algorithm 3 `ResourceAgent` behaviour, showing how a `ResourceAgent` treats each type of request it can receive.

```
initializeState()
loop
    Request ← retrieveRequests()
    if Request == WHAT-IF then
        return WHAT-IF-ANSWER
    end if
    if Request == INTENTION then
        intentionQueue.add(INTENTION)
    end if
    if Request == SERVICE-REQUEST then
        update averageExecutionTime
    end if
    if Request == DROP-PHEROMONE then
        incrementPheromoneLevel()
    end if
end loop
function incrementPheromoneLevel
    pheromoneLevel ← pheromoneLevel + pheromoneDelta
end function
function decrementPheromoneLevel
    pheromoneLevel ← pheromoneLevel − pheromoneDelta
end function
function initializeState
    intentionQueue ← Empty
    averageExecutionTime ← 0
    pheromoneLevel ← 0
    pheromoneDelta ← 0.001
end function
```

4.2.3 ExplorationAnts

The main responsibility of an `ExplorationAnt` is to create an execution plan for a service composition. This plan is made by assigning a component service to each activity from the composite service graph.

An `ExplorationAnt` achieves its goal of creating a plan by crawling the service network and finding good candidate component services to be used in a service composition. `ExplorationAnts` are also responsible for finding not only, any candidate component services, but the most suitable ones, according to the objective of the composite service, which normally is to reduce the completion time of the entire composition.

An `ExplorationAnt` is created with a composite service description graph, and an objective function. The composite service description is used to inform which types of activities are needed by a composite service and in which order they are needed. The objective function is used to instruct the `ExplorationAnt` on how to evaluate the alternative candidate services. The composite service description and the objective function are informed by the `TaskAgent` that creates the `ExplorationAnt`. Each node from the graph corresponds to an activity that has to be performed by a component service. The `ExplorationAnt` searches for component services that can fulfil each needed activity traversing the graph, formed by `ResourceAgents`, and evaluating the possibilities at each node.

Having the information about which component services are needed by a composite service, an `ExplorationAnt` searches the network looking for good candidate component services. A particular problem is how to evaluate a large number of candidate component services for each activity, in a short time. This is particularly challenging, since we are working with service networks of thousands of nodes. An `ExplorationAnt` performs a probabilistic search on the service network and every time it finds a new candidate service, it chooses which one to use based on two attributes: the pheromone information, and the heuristics information.

An `ExplorationAnt` searches the service network by communicating with `ResourceAgents` which are associated to composite services that can provide services for the needed activities. In order to fulfil their goals, `ExplorationAnts` look for `ResourceAgents` that: (i) offer the required type of services, (ii) can perform the operations at the required time, and (iii) have good quality, QoS.

Every time an `ExplorationAnt` has to evaluate which path to follow, that is, decide which `ResourceAgents` to engage, it constructs a probability table with entries for each candidate component service and an associated probability. The probabilities are created using the pheromone levels and an heuristic function η

given by the `TaskAgent` which created the `ExplorationAnt`.

The function η is evaluated every time an `ExplorationAnt` checks for the quality of a service provided by a given `ResourceAgent`. Every time an `ExplorationAnt` moves through the network, it has to decide which new `ResourceAgent` it should engage with, as illustrated in Figure 4.5.

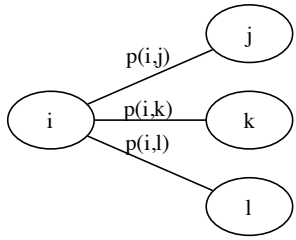


Figure 4.5: At every `ResourceAgent` i an `ExplorationAnt` assigns probabilities for every following `ResourceAgent`. In this example, an `ExplorationAnt` jumps from the `ResourceAgent` i to the `ResourceAgent` j with probability $p(i, j)$.

At each exploration step, an `ExplorationAnt` constructs a probability table, which assigns a certain probability to engage with each possible `ResourceAgent` at that point, according to the Equation 4.1.

$$P_{ij}(t) = \frac{[\tau_{ij}(t)]^{\alpha}[\eta_{ij}(t)]^{\beta}}{\sum_{l \in N_i}[\tau_{il}(t)]^{\alpha}[\eta_{il}(t)]^{\beta}} \tag{4.1}$$

where τ is the pheromone level, α indicates how worth is the pheromone information to the `ExplorationAnt`, η is the heuristic that takes the QoS into account, and β indicates how worth is this quality information to the `ExplorationAnt`.

The α and β parameters, thus, have a high impact in the outcome of the plans created by `ExplorationAnts`. The combination of the α and β values determine how long it will take for a large number of ants to converge to a certain set of component services. For instance, performing a sensitivity analysis, we see that in the case where $\alpha - \beta = 5$ `ExplorationAnts` will mainly follow the paths with the highest pheromone concentration. In other words, the `ExplorationAnts` will only select `ResourceAgents` which were previously selected and were good for other compositions. Another aspect of a high α value is that new `ResourceAgents` will have a very small probability of being selected by the `ExplorationAnts`, and will take a very long time for good quality `ResourceAgents` to start being selected.

On the other hand, in the case where $\beta - \alpha = 5$ the current QoS of a `ResourceAgent` will have much more weight in the probability table of the `ExplorationAnt` than any past experiences with that particular `ResourceAgent`, having a much higher chance of being quickly selected.

After constructing this probability table, the `ExplorationAnt` generates a random value between 0.0 and 1.0 (according to the uniform probability distribution) and selects the corresponding `ResourceAgent` from the probability table.

`ExplorationAnts` only explore the service network for a limited amount of time, which is indicated by the *exploration timeout* of that particular `ExplorationAnt`.

After, either reaching their *exploration timeout*, or finding a set of component services which fulfil all tasks needed by a composite service, the `ExplorationAnts` return a plan to their originating `TaskAgent`. It is important to note that the coordination mechanism operates an order of magnitude faster than the service operations. This difference in execution time means that the information stored at the coordination layer converges very fast as well.

In summary, an `ExplorationAnt`'s goal is to find suitable candidate component services to fulfil all activities of a composite service. `ExplorationAnts` have three main parameters, which are: i) α, indicating the weight given to the pheromone information, ii) β, indicating the weight given to the heuristic (quality information), and iii) *exploration timeout*, a time in milliseconds, that the `ExplorationAnt` has to explore the network, before returning its results to its master `TaskAgent`.

4.2.4 Intention Ants

An `IntentionAnt` responsibility is twofold: i) share the intentions of a `TaskAgent` to use a set of component services, ii) give feedback to the system about the quality of the component services it has selected. The intention sharing is crucial to the proper working of our approach. It is via intention sharing that other `TaskAgents` can indirectly coordinate their actions and, ultimately, decide to engage or not with a particular component service.

We model an intention as a tuple $I = (ta_i, ra_j, t_k)$ which indicates that a `TaskAgent` (ta_i) plans to use a `ResourceAgent` (ra_j), at time t_k. An intention is not a reservation to use a resource, but simply an indication of a high probability to use a particular resource at a given time. As such `TaskAgents` have to be always prepared to select other resources, in the event the component services

they intended to use are not available at the desired time. `IntentionAnts` spread this intention information to all `ResourceAgents` present in the `TaskAgent` plan.

The second responsibility of an `IntentionAnt` is to give feedback information to the system, regarding the quality of the component services used by a `TaskAgent`. `IntentionAnts` do this by drop pheromones about the quality of the execution of services provided by each `ResourceAgent` they have used so far.

The pheromones, in our model, represent quality information about the past execution of component services used by different `TaskAgents`. The information contained in the pheromones indicate which `ResourceAgent` was used, when it was used, and how accurate were the quality predictions given by the `ResourceAgent`. A pheromone has a fixed value and indicates to which component service it refers to. The more ants dropping pheromones about a particular component service, higher the probability that other ants will use that service.

It is interesting to notice, that a system has thousands of `ExplorationAnts` and `IntentionAnts` working at any moment, but they never directly communicate. `IntentionAnts` and `ExplorationAnts` only communicate with the `TaskAgent` which created them and with `ResourceAgents`, which facilitates the design of the system. Remembering that `ExplorationAnts` make use of this pheromone information to make better decisions about which component service to select for their compositions.

4.2.5 Pheromone Evaporation

The longer the system is working, the higher the pheromone information accumulated in each node in the system. If a pheromone trail becomes to strong, `ExplorationAnts` will very likely follow this trail instead of exploring new solutions. Completely avoiding the exploratory behavior is dangerous, both for natural ant colonies as well for artificial ones, since the system can be stuck to sub-optimal solutions and will lack prompt alternatives in the event of failures.

To avoid the problems of being stuck to only one certain path, we use a mechanism called **Pheromone Evaporation**. This mechanism constitutes of decreasing the pheromone level associated to a certain path as time pass by.

It is modelled as a function $f(t, p, o) \leftarrow p * e^{-t/o}$, where t is the elapsed time, p is the current pheromone level, and o is the exploration timeout of the `ExplorationAnts` in the system.

By evaporating pheromones, we know that if a component service is not used for a certain time, even if its initial pheromone level is very high, the pheromone level will vanish after a while. We also guarantee that the evaporation frequency happens at the same order of magnitude of the exploration timeout of `ExplorationAnts` in the system, what avoids loosing the information too quickly or maintaining very old information.

4.2.6 A Note on `TaskAgents` Commitment Strategy

When an agent has a plan and does not ever reconsider its plan, this agent is said to be strongly committed to its plan, or to have strong commitment. On the other hand, an agent which defines a plan, but keeps reconsidering such a plan, that is, keeps looking for alternatives to its plan, is said to be weakly committed.

Due to the dynamic nature of the environment on which services, and its agents, operate, changes in availability, quality, etc, can be very frequent. A network connection can fail, a computer node can physically break, an operating system can havoc, or a DoS attack can be targeting one or several component services. In such environments, both extreme commitment cases help to create systems that can very easily be trapped in local maxima, minima, or even unfeasible solutions.

For instance, a strongly committed agent can create a plan to use a number of services, which become unavailable a few milliseconds after the plan's creation. Such strongly committed agent would halt the execution of its composite service until the selected services to participate in the composition would become available again. It is clear that if one of the goals of the agent is to minimize the execution time of the service composition , for instance, a strong commitment can lead to bad solutions.

A weakly committed agent can lead to bad solutions as well, but because of not being able to select a component service to participate in its service composition. A weakly committed agent creates a plan, select component services to participate in such plan, but constantly reconsider its selection before allowing its composite service to request an operation to a component service. This commitment strategy becomes a problem if the rate change on the environment is such that leads the agent to spend more time deciding which plan to follow than just following a plan taken before.

The main problem of deciding how committed an agent should be is that being strongly or weakly committed, also called an agent's commitment level, can be good or bad in terms of an agent's goals, depending on the particular rate of

changes from the environment and how fast the agent can compute its plans. Another problem to consider is that the different commitment strategies from each agent can have impact on the overall quality of a MAS system. Perhaps locally, being weakly committed can be achieve good solutions from the point of view of an individual agent, in the short term, however, it can bring disastrous consequences to the MAS system as a whole.

In our approach is decided to have a intermediate commitment from the `TaskAgents`. They continue exploring alternative plans, via their `ExplorationAnts`, but only change their intentions if the reduction in the execution time is more than 80% of their current plans. Otherwise the `TaskAgent` simply follows its previous plans.

Chapter 5

Evaluation

This chapter discusses the experiments we performed to evaluate our algorithms and their results. We start by depicting our hypotheses and the experiments we performed to test them. We explore the dimensions that can bring a better understanding of the trade-offs of creating a large scale service system.

An alternative mechanism for creating service compositions in large scale networks is to create a purely reactive selection mechanism. A reactive mechanism only searches for candidate component services when they are needed and select one according to a desired objective function. We perform experiments with a purely reactive solution, explained below, to provide a benchmark for the Delegate MAS mechanism.

In the experiments, we mainly use the following metrics:

- **Composition time**. Is the time duration it takes for a service composition: i) to search for candidate component services; ii) to select a component service; ii) to execute the desired operation of the component service; For all the tasks in the composite service.
- **Communication Cost**. Is the number of messages exchanged between all the services participating in the system.
- **Fairness in resource allocation**. Measures the distribution of tasks between the resources available in the system.

It is relevant to note that services may be geographically dispersed. The locality of the services is taken into account by the **Composition time** metric, since

this metric takes into account the time it takes to search for a service as well. This search time mirrors the distance (in network delays) between the services.

5.1 Hypotheses

We want to clarify whether it is possible to efficiently create dynamic service compositions in a completely decentralized fashion, especially in the presence of failures. Based on state-of-the-art research, we elaborated three hypotheses that, we believe, are relevant to understand large scale and decentralized service systems.

Hypothesis 1: The scale of the network does not affect the composition time of the created service compositions.

Efficiently managing the ever growing number of nodes interacting on cloud datacenters is challenging [27, 42]. Systems have to efficiently use computing, network, and cooling resources. Even more challenging is the fact that the service systems we envision have to scale beyond datacenter boundaries. Thus we need to test if our algorithms can support a growing number of interacting services and still provide reasonable composition times for the service compositions.

Hypothesis 2: Using Delegate MAS algorithms lowers the variance of service composition times, compared to purely reactive solutions.

There are use cases, such as in video streaming applications, where it is more important to have a given certainty regarding the service composition times and cost, than to have optimal ones. A composite service owner may prefer that all clients from his composite services receive the same, expected, QoS, than having a few composite services with high quality and a large number having poor or unexpected quality.

Hypothesis 3: A system using Delegate MAS gracefully degrades on the presence of very large failures.

The environment where services operate, be it cloud providers or datacenters spread around the world, is very dynamic. Failures are a constant in such

environments. Besides being interested in small failures, or small scale disruption, which we define as 20% of component services failing, we are interested in testing if our solution can cope with large scale failures, or large scale disruption, which we define as 80% of component services failing.

We assume there is a high replication of component services. Our hypothesis is that large scale failures will not disrupt the system. We check if the system still operates in the presence of large scale failures and how well it performs under such events.

5.2 Experimental Setup

We want to understand the system wide behaviour of our algorithms executing on a large scale service network. In order to create this understanding we developed a prototypical distributed system that implements the abstractions from the `DMAS` model.

The benefit of performing experiments with a real distributed system is that the experiments can expose unforeseen events. Such events can be issues with our model's protocols, protocol interactions, and most importantly, the effects of the parallel and independent execution of each service and service manager. We implemented our `DMAS`'s model abstractions and deployed our system on a computer cluster.

In order to test each hypothesis we performed several experiments. Each experiment was composed of a scenario, metrics, and analysis of the results. The services are deployed on a cluster of 32 nodes. Each node has 12 microprocessors with hyper-threading, totalling 24 execution threads per node. Each node has 96 Gb of RAM. The node's operating system was a Linux with kernel 3.2.0-52-generic SMP.

The prototype is a real distributed system, with services exposing their operations on the network and being remotely invoked as well. Service's operations are simulated by generating a random time which represents how long the computer will be busy. Our component services are implemented to simulate the duration of operations of real services. Figure 5.1 depicts the different levels of abstraction used in the prototype. Services are implemented in the form of actors, so that the implementation does not need to care about marshaling/unmarshaling of messages, etc. Each actor, is deployed on a Java Virtual Machine (JVM), and each JVM is assigned to one core of a node.

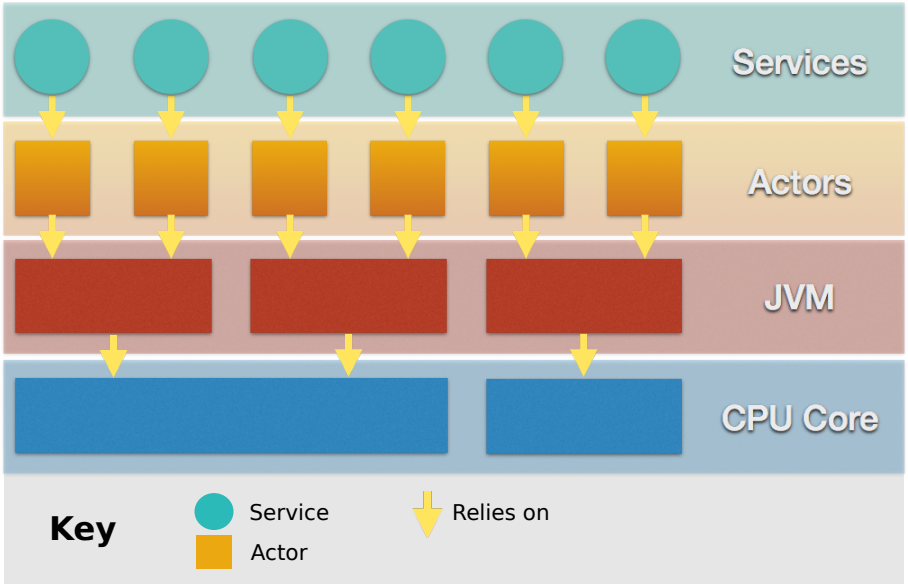


Figure 5.1: A service, in the prototype, provides different operations to other services. A service is realized as an Actor which provides the mechanisms to serialize/de-serialize message payloads, etc. Small actor groups are deployed on a Java Virtual Machine (JVM) which is assigned to a particular CPU core.

5.2.1 Scenarios

We are interested in the behaviour of our algorithms under different scenarios which represent situations that can be faced by real systems deployed on the internet.

An experimental scenario has the information about the network topology information and the configuration used for the experiment. The network topology defines how a particular service is connected to the other services in the system. Each service is only aware of the presence of a small number of other services, given by the topology of the network. In order to make our experiments more realistic, we used internet topology data collected by the CAIDA project. The CAIDA Skitter project created an internet topology graph after running traceroutes from scattered sources to millions of internet hosts, in the year of 2005 [64]. Figure 5.2 shows a sample network containing 1000 nodes created using the Skitter data. We use this topology data to indicate to each service in our system, which other services they are directly connected to. That way, we avoid relying on any artificial central registry in our experiments. Each

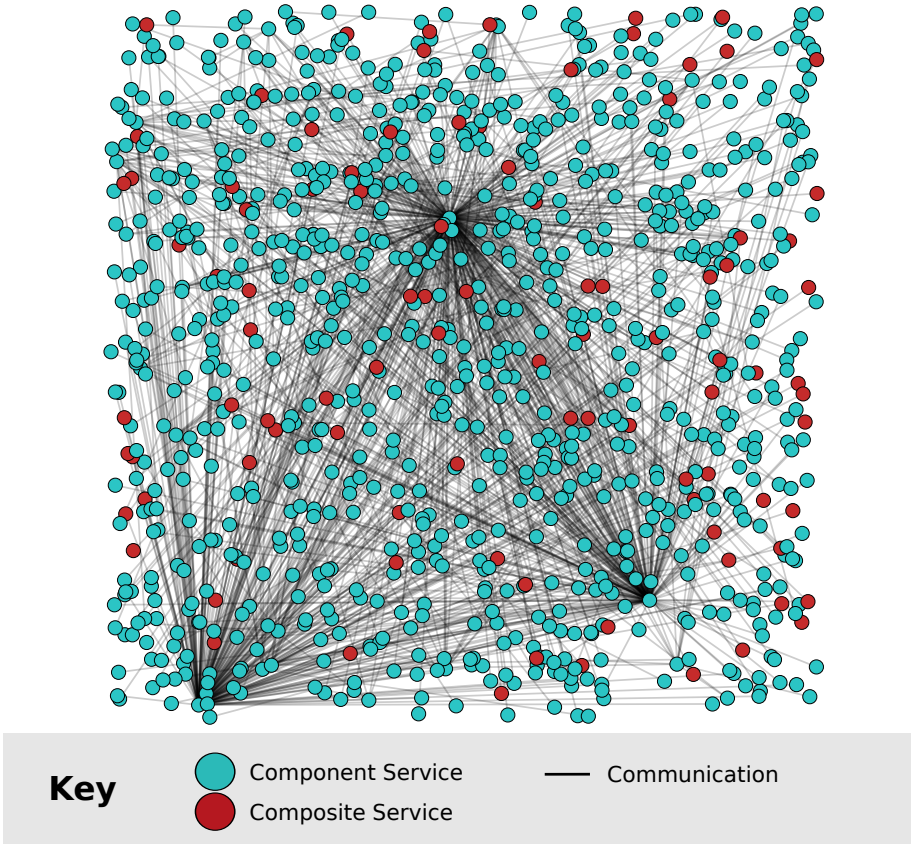


Figure 5.2: Each node in the graph represents a service. There are two service natures, composite and component services. The edges represent to which other services, a service is directly connected.

node represents a service, either a Composite or Component service.

The configuration used for the experiments has information about each service's parameters, such as execution timeouts, and information about the failure that should happen in the experiment. The failures configuration specifies when the failures should start happening in the experiment, the probability for a failure to happen and the average failure duration.

We assume a fixed ratio between composite and component services, no matter the size of the network. We also assume that the number of composite services

to be much smaller than the number of component services. The composite services used in our experiments have either two or three activities. We decided to use a small number of activities per composite service in order to still be able to understand and analyse the behaviour of the system. Each composite service is instantiated with a linear workflow containing either 2 or 3 activities. Workflow activities can be of three different types A, B, C. Component services only offer one type of service, either $A, B,$ or C

Service types are assigned to component services using the uniform probability distribution, meaning that it is expected that there will be 1/3 component services of each service type. The time for the execution of a task by a component service is randomly generated, according to the uniform probability distribution, with an average 20 seconds plus the number of requests queued on the component service. Table 5.1 depicts the number of composite and component services of which types and their average time to process a request.

Network Topology

Each network size was created using the same data from the CAIDA project. We can see, at Table 5.2, that a few network metrics change from a network size to another, such as the average degree, which is shows the average number of edges connected to a node, in the network with 64,0001 nodes. Such differences show that the number of nodes is not the only relevant information regarding each network size, but how the nodes are connected is also relevant and can impact the algorithms we test. Table 5.2 provides a summary of the networks we have used in our experiments.

Each service in the experiment is configured with a fixed execution timeout. We empirically tested several timeouts that would improve the overall resource usage of the prototype with the given scenarios and fixed the execution timeout to 1.95 seconds for any service in the system.

5.2.2 How to Evaluate DelegateMAS

Nowadays there are no studies, as far as we are aware, with real large scale service systems as large as we study. Our scenarios' large scale makes it unfeasible to look for optimal solutions for the service composition problem. Besides being computationally expensive to find optimal solutions, we believe there is no sense in using centralized algorithms for a problem that is decentralized by nature. The main risk of trying to create optimal centralized solutions for

Table 5.1: Characteristics of the Services used in our experiments

Scenario	Service Type	Nb. Services	Mean Exec.Time (s)	StdDev (s)
250	Composite 2	14	-	-
	Composite 3	13	-	-
	Service A	75	19.41	5.65
	Service B	79	20.23	5.88
	Service C	70	19.88	5.78
1k	Composite 2	62	-	-
	Composite 3	34	-	-
	Service A	296	19.49	5.42
	Service B	331	19.84	5.72
	Service C	278	20.16	5.53
4k	Composite 2	271	-	-
	Composite 3	160	-	-
	Service A	1217	20.1	5.77
	Service B	1194	19.82	5.7
	Service C	1159	19.88	5.73
16k	Composite 2	1029	-	-
	Composite 3	494	-	-
	Service A	4715	20.03	5.82
	Service B	4762	19.91	5.79
	Service C	5001	19.90	5.77
64k	Composite 2	4327	-	-
	Composite 3	2129	-	-
	Service A	19228	20.03	5.76
	Service B	19270	19.99	5.79
	Service C	19047	20.02	5.73

a decentralized system is that the optimal solver may use stale information, generating irrelevant solutions.

Given the difficulties and problems of using optimal solvers, we evaluate our solution, by comparing Delegate MAS to a purely reactive approach, explained below.

We performed each experiment multiple times to diminish the influence of randomness in our results. Each experiment having large scale scenarios, more than 10.000 nodes, was performed 10 times. For smaller scenarios, we performed each experiment at least 30 times.

Table 5.2: Service Network Metrics

Metric	Value	Value	Value	Value	Value
Nb. Nodes	251	1001	4001	16,001	64,001
Nb. Edges	250	1034	4034	18,752	103,101
Diameter	2	4	4	5	6
Radius	1	2	2	3	3
Avg. path length	1.99	3.09	2.53	3.74	7.25
Nb. shortest paths	62750	1001000	16004000	256016000	2147483647
Avg. Degree	1.992	2.066	2.016	2.344	3.222
Modularity	0.0	0.749	0.350	0.779	0.749
Nb. Communities	1	30	23	33	14
Avg. Cluster. Coef.	0.0	0.043	0.043	0.271	0.1
Total triangles	0	3	3	942	12014

Reactive Service Composition

We created a purely reactive service composition mechanism to compare to our Delegate MAS solution. The reactive algorithm is executed by a Service Manager, every time there is a new execution of a service composition. When there is a new service composition instance, the Service Manager searches the network, looking for suitable component services for each task of the composite service. The Service Manager searches, for a limited time, for at least five component services capable of performing the desired task. After finding the five component services, or having a timeout, the Service Manager evaluates the component services' QoS and selects the best one. After selecting a component service, the Service Manager indicates the component services's address to the composite service and waits for the execution of the needed task.

The Service Manager repeats this process of looking for component services, selecting one, and waiting for the execution of an operation while there are still not completed tasks in the composite service.

The main aspect of the reactive service composition is that there is no plan of which component service to use in the future, or any coordination between the several Service Managers in the system.

5.2.3 Performing Experiments

In order to test each hypothesis, we perform experiments using both Delegate MAS and the reactive in different network sizes. An experiment consists of

multiple runs of the system, having exactly the same configuration per run but with different seeds for the random number generators used by the agents.

Each run consists of the following steps:

- Deploying the Composite and Component services, following the scenario for a particular network scale, on the cluster.
- Triggering the Composite Services to starts their service composition, so that composite services can start looking for candidate component services to participate in their service compositions.
- Annotating the time it takes for each composite service to execute all its activities, that is annotating the **composition time**.
- Annotating the messages exchanges between the services, so that we can learn how much information is needed by each mechanism.

We perform the experiments using the Delegate MAS approach and then using the reactive approach for each network size, so that we benchmark our approach against the reactive approach. We assume that variations on each experiment run follows the normal probability distribution, thus we execute each experiment multiple times, and calculate the errors for a 95% confidence interval.

5.2.4 Handling Failures

We generate failures by sending Failure messages to each component service. However we never send Failure messages to component services which are already participating in a service composition. Another limitation of our experiments is that we do not take into account the possibility of correlated failures. More specifically, we do not model failures that affect a cluster of services and immediate composite services which are close to the cluster.

5.3 Hypothesis 1: The scale of the network does not affect the composition time of the created service compositions.

We want to test if the scale of the network, that is, the number of services interacting over the network, has any impact over the composition time of the service compositions. We test this hypothesis by creating scenarios with

different network sizes, but having the same topology and the same ratio of composite services per component services. The main aspects we are interested in evaluating for this hypothesis are:

- Composition time. Are the composition times affected by network size?
- Fairness in resource allocation. Is the system capable of fairly allocating resources in different network scales?
- Communication costs. Can the communication costs of our solution hinder applying it to very large scale networks?

5.3.1 Experiments Results for Hypothesis 1

We are interested in how well our Delegate MAS solution performs compared to the reactive solution, in very large networks. For that end, the first criterion we use is the average composition time for the completion of the composite services. Figure 5.3 shows the average composition time distributions obtained by the composite services, for different network sizes. It also shows how the network size affects compositions having 2 or 3 activities.

It is clear from Figure 5.3 that the Delegate MAS solution is capable of creating service compositions which are good even with the increasing size of the network. We can see that our solution creates service compositions which take less than 65% of the time to execute, compared to the compositions created by the Reactive approach. The difference in quality of the created compositions is even greater in large scale scenarios, having more than 16k services. In a service network having 16k services, a service composition having three tasks took an average of 73.95s $\pm$ 1.17s, using the reactive approach, and 60.91s $\pm$ 2.02s, using the Delegate MAS approach. The difference in the quality of the compositions between the two approaches, for 16k services network and compositions having three tasks, was on average 13,04s, which accounts to the Reactive approach creating compositions which took more than 21% time to execute than our approach.

If we look at the results for very large networks, having 64k services, the difference in the quality of the created compositions was even greater. The Reactive approach created compositions (with three tasks) which took on average 94.26s $\pm$ 3.53s, while our approach created compositions which took on average 59.03s $\pm$ 1.89s. This result was surprising, since the Reactive approach compositions took 35.23s, or 59% more time than our approach. Our solution always outperformed the purely reactive approach in the studied scenarios.

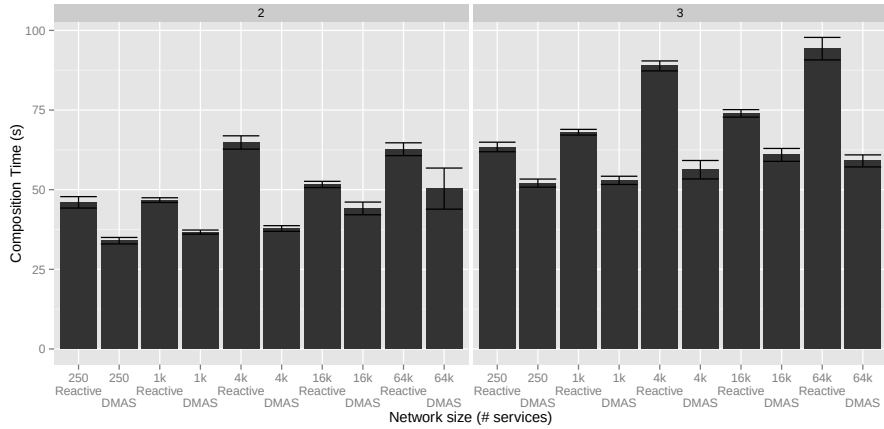


Figure 5.3: The bigger the network size, the better are the composition times. This happens because, each Composite Service has a higher probability of finding better component services, since there are more component services to be searched. The graph also shows that there the network size has a bigger effect on compositions with more activities (3), than on compositions with only 2 activities.

Another interesting aspect concerns the stability of our approach compared to a purely Reactive approach. Figure 5.4 shows that the composition time of the compositions created using Delegate MAS is only slightly affected by the number of services participating in the service network. We can see, for instance, that the composition time increases 25% when the number of services in the network increases 400% (from 16k services to 64k services).

Next, we investigate if there are any substantial differences in terms of communication overhead for different network sizes. Figure 5.5 shows the communication costs for both, reactive and Delegate MAS approaches for different network sizes.

Figure 5.5 shows that both approaches need to exchange a large number of messages to complete the service compositions. However, for the network having 64k services, Delegate MAS needed to exchange 2.6 times more messages then the Reactive approach. On the other hand, our approach produces service compositions which are on average 1.6 times shorter than the compositions created by the Reactive approach. The trade-off a system designer faces concerns the communication costs versus the quality each approach produces.

We are also interested in the distribution of the composition times between the

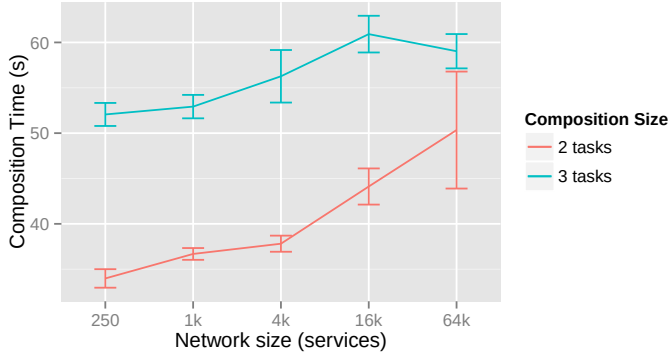


Figure 5.4: The quality of the compositions created using Delegate MAS is not affected by the number of services participating in the service network.

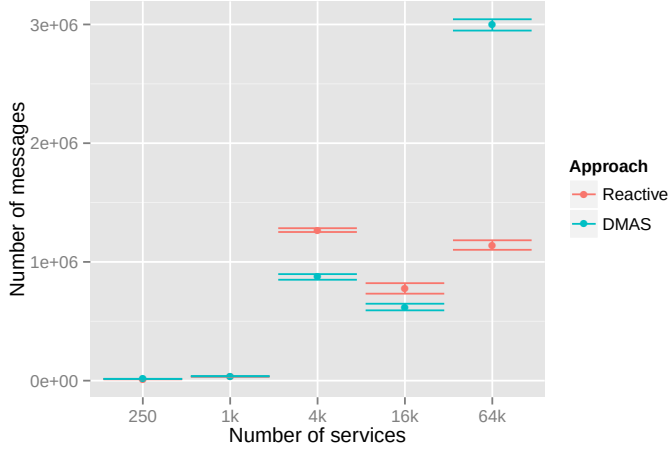
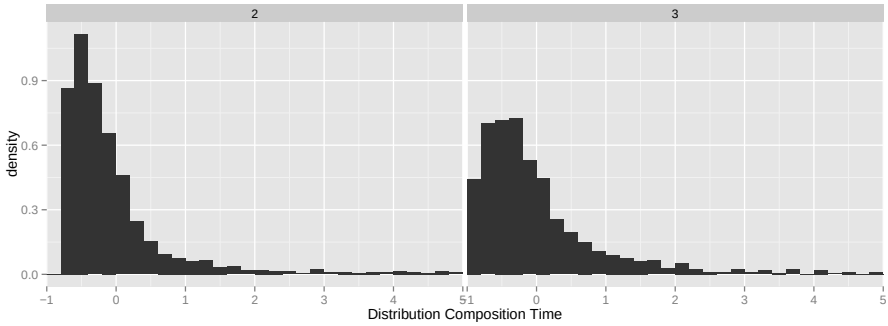
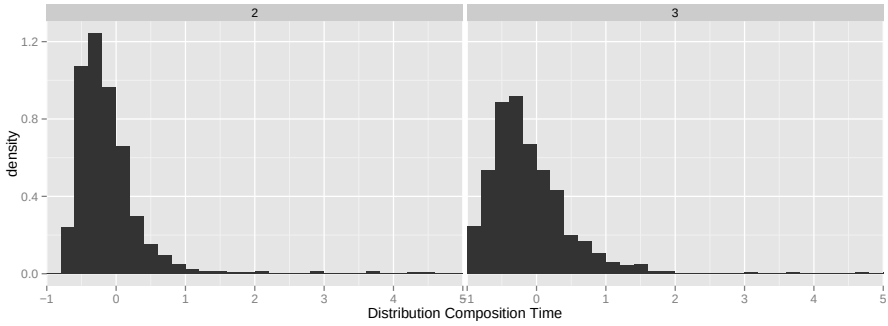


Figure 5.5: Communication costs of a Reactive and Delegate MAS approaches for different network sizes.



(a) Reactive Approach, 16k services

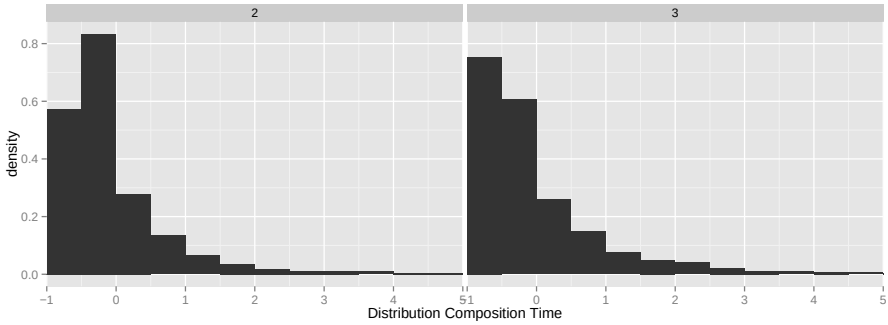


(b) DMAS Approach, 16k services

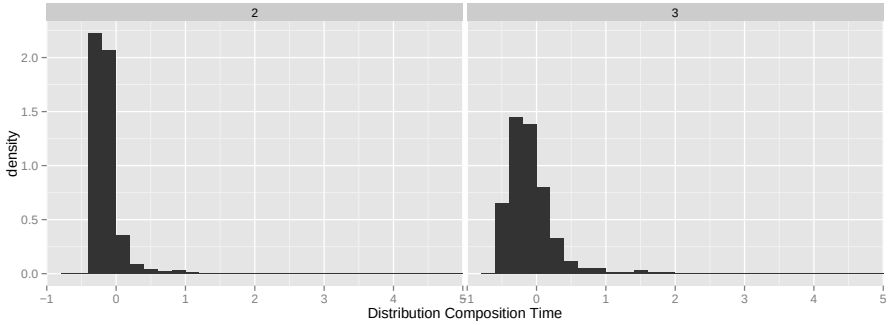
Figure 5.6: The distribution of composition times for compositions shows how fair the mechanism can be in allocating component services to composite services, avoiding having too many composition which take too long to execute.

many composite services. A distribution with a very long and fat tail means that the algorithms are not fair, and are allocating more resources to a small group of composite services, while there are composite services starving for resources. Simply showing the variance is not enough to learn the percentage of services having bad composition times. Ideally the distribution of composition times will have a high peak around the mean, having the majority of compositions executing around the average composition time. In order to to this analysis we normalize the distribution of composition times, and check how spread the compositions are. We perform this analysis for networks having 16k and 64k services.

It is possible to see from Figures 5.6, and 5.7 that there is a small difference in



(a) Reactive Approach, 64k services



(b) DMAS Approach, 64k services

Figure 5.7: The distribution of composition times is much more spread in the reactive approach, showing that there are more services taking longer to execute, while the distribution is much more concentrated around the average in the Delegate MAS approach. The histogram indicates that service compositions using the Delegate MAS have a smaller range of composition times, which indicates they better share the available component services.

the distribution of composition times between the two approaches. In both cases, Delegate MAS is more centered around the mean than the reactive approach.

5.3.2 Conclusion Hypothesis 1

In this subsection we have tried to falsify our hypothesis that our approach was capable of creating good quality compositions in very large scale service networks. We performed experiments with networks varying from 250, to 64 thousand services and analysed both the quality of the created compositions and the communication costs incurred to create such compositions. We concluded that our approach was capable of creating better quality compositions, in terms of composition time, at a higher communication cost than a purely reactive approach. This shows that there is a trade-off between the quality of the created compositions and the communication cost to create them, which , to our understanding, can not be avoided.

We also show that Delegate MAS properly scales even with an exponential growth in the size of the network where it is executing. However, there is a high communication cost in the 64k network.

5.4 Hypothesis 2: Using Delegate MAS algorithms lowers the variance of service composition times, compared to purely reactive solutions.

Many times it is more important to have better estimates for how long a service composition is going to take, than to minimize its completion time. In a video streaming system, for instance, it is more desirable to guarantee that almost all users have the same user experience, than to completely optimize the system in order to make it better for a few users and worse for other users.

A way to measure this quality is by measuring the variance of composition times. Given service compositions which use the same types of services, it is desirable that they would take the same amount of time to complete. We evaluate our approach regarding the variability of the composition times, and test if it can minimize the variance of composition times, compared to a purely reactive solution.

We expect that our approach, can find suitable component services that will allow the creation of compositions which do not have a high variance in composition times. We test this hypothesis by performing experiments with service networks

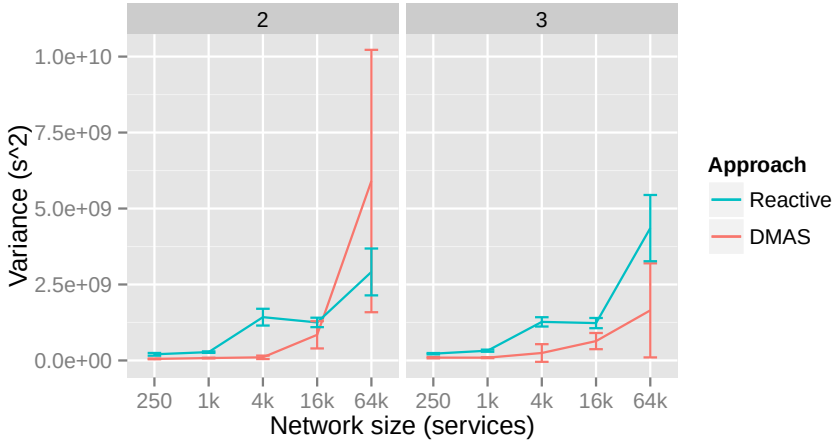


Figure 5.8: Analysis of the variance of composition times for our approach and for a reactive approach.

having different sizes. The networks have the same number of the networks described in Table 5.1 (Scenario 16k, and 64k respectively) and have the same interconnection properties as described in Table 5.2.

We are mainly interested in the following attributes:

- Variance of the composition times. Can our approach create compositions with a lower variance in their composition times?

5.4.1 Experiments Results for Hypothesis 2

We performed 10 runs per network size, and analyse the results we have found. Figure 5.8 shows our analysis for the variance of the composition times of our approach versus the reactive approach.

It is interesting to note that our approach (Delegate MAS) is consistently better, that is has a lower variance, for almost all network sizes. Our approach manages to have a lower variance for compositions having 3 activities in all networks. However, the reactive approach manages to create compositions having 2 activities, with a lower variance in networks having 64k services. Our approach performed poorly in service compositions having 2 activities in this very large network (64k services). We believe this can be attributed to Delegate

Table 5.3: Summary of the variances of composition times in Delegate MAS and a Reactive approach.

Approach	Median	Mean
Reactive	12.40e+8	13.47e+8
DMAS	1.74e+8	9.69e+8

MAS minimization of composition times, which was consistently better than the composition times created by the reactive approach, as shown in Figure 5.3. Thus, we believe, that in order to achieve such low composition times, a number of services had large composition times (compared to the best ones), leading to a high variance in this scenario.

If we ignore the network sizes and analyse the averages of the variances, we can see that our approach has lower mean variances as well. Table 5.3 shows summary statistics for the reactive and our approach.

We apply the *Vargha-Delaney A measure* statistic to evaluate whether there is a relevant difference by using the Delegate MAS and the Reactive approach in this particular experiment. The *Vargha-Delaney A measure* communicates how often one particular technique outperforms the other [107]. The A-measure calculated for Delegate MAS and the Reactive approach is 0.72. This means that, according to this statistic, our approach obtained better results 72% of the time, while the Reactive approach manage to have better results only in 28% of the time.

5.4.2 Conclusion Hypothesis 2

Having a 95% confidence interval, our approach was consistently better than the reactive approach for compositions with 3 activities. However, to our surprise, our approach performed poorly in service compositions having 2 activities in a network having 64k services.

Our approach consistently created better service compositions than the reactive approach, what is also indicated by the results of the *Vargha-Delaney A measure*. Considering that our approach performed worse than the reactive approach, only in one scenario, we conclude our hypothesis to be valid and that our approach can, indeed lower the variance of composition times.

5.5 Hypothesis 3: A system using Delegate MAS gracefully degrades on the presence of very large failures.

We configure the algorithms Delegate MAS and the Reactive mechanism with the same timeout to explore the network. The Delegate MAS algorithm is configured with $\alpha = 0.5$ and $\beta = 5$, (eq. 4.1), which means that it gives more preference to not known component services (heuristic information), than to the previous quality offered by a known component service (pheromone information).

5.5.1 Failures

We are interested in the behaviour of the system in the occurrence of failures. We focus on failures which affect 20% and 80% of the services in the system. A failure which affects 20% of the services is called a small scale disruption. A failure which affects 80% of the services is called a large scale disruption. We simulate a failure by having a special type of message, called a "Failure Message", that triggers a fail state in a service. When a service is in a fail state, it will refuse any connections from other services.

Initially we let the system execute for 10 seconds, then we start sending failure messages to the services in the system. When a service receives the "Failure" message it stays on average 30 seconds in a fail state. In this experiment, it means the service will remain in a fail state until the end of the simulation.

We are also interested in the effects of failures in large scale and very large scale systems. In a small scale disruption, for each service on the system, we generate a random number between [0,1] using the uniform probability distribution, if the number is smaller than or equal 0.20, we send a "Failure" message to that service. For the large scale disruption, we follow the same procedure, but send a "Failure" message if the generated number is smaller than or equal to 0.8.

5.5.2 Experiments Results for Hypothesis 3

We are interested in how well the reactive mechanism and the Delegate MAS work in a large scale network with failures. In order to measure how the system performs, we mainly focus on the composition time.

An important aspect of creating service compositions is to guarantee that the system operates properly during the entire execution of the system. We are interested in many aspects regarding the system behaviour, when services suddenly fail. We also want to investigate how good a service composition is in relation to the remaining service compositions, for instance, measuring the standard deviation of the composition execution times, in the presence of failures.

We begin describing the system behaviour when it is subject to a small scale failure, that is, 20% of the nodes are failing. We describe the system for a service network having 16k services and for a bigger network, having 64k services.

Directly comparing both approaches, we can see that a purely *Reactive* approach creates compositions with higher average composition times than the Delegate MAS approach. Figure 5.9 shows a comparison of the averages of the composition times created by the two different approaches, in networks having 16k and 64k services. It is interesting to note the difference in the average composition times for composite services having 2 or 3 tasks. In a network having 16k services, Delegate MAS is 4.55 % better than the *Reactive* approach when the composite service has only 2 tasks. On the other hand, Delegate MAS is 24.15 % better than the *Reactive* approach when the composite service has 3 tasks.

The behaviour of both algorithms under 20 % failures in a very large network, having 64k services, is not as efficient as in a smaller network. The reactive approach for compositions having 2 tasks was only 2.36 % slower than Delegate MAS. Again, in a network having 64k services, Delegate MAS performed better for compositions having 3 tasks. In compositions with 3 tasks, the *Reactive* approach was 4.6 % slower than Delegate MAS.

Figures 5.10 and 5.11 show the normalized distribution of the composition times of the created service compositions, in a scenario with 20% of nodes failing and 16k services, for the *Reactive* and Delegate MAS approaches respectively. It is possible to see in Figure 5.11 that the distribution of the average composition times in the Delegate MAS approach is more concentrated around the mean, than the purely reactive approach. Another interesting aspect is how Delegate MAS is more affected by the different number of tasks in a composite service. In Delegate MAS the distribution of composition times around the mean for composite services having 3 tasks is larger than for composite services with only 2 tasks, as illustrated in Figure 5.11. We believe this difference is due to the fact that composite services with more tasks may be more affected in the presence of failures, since they rely on more component services.

Networks having 64k services and 20% failure have a distribution of composition times which is very similar to the distributions for networks having 16k services.

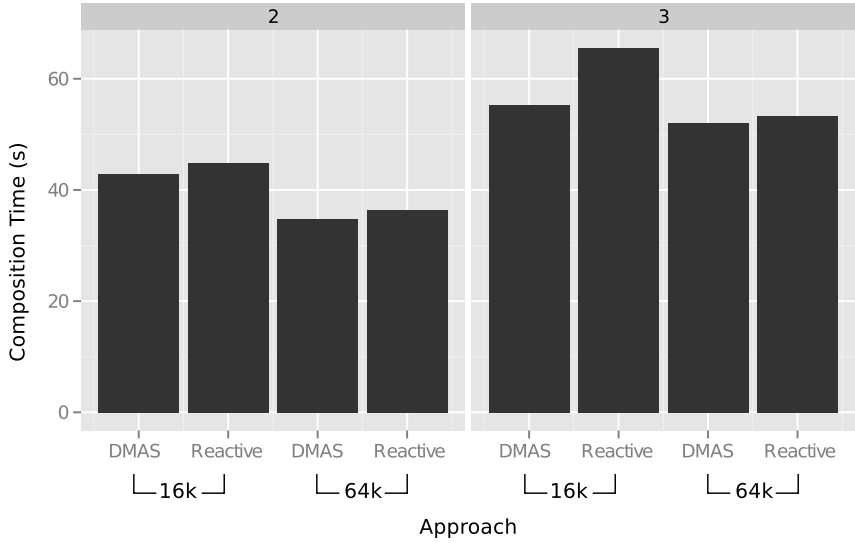


Figure 5.9: Comparison of the average composition times of service compositions created using Delegate MAS or the Reactive approach. The service compositions are made of 2 or 3 tasks, and the system was evaluated with a 20 % component service's failure rate. Delegate MAS selects component services which produce a better (shorter) composition time for the different network size and composition types.

Hence, we do not show these results here.

We are also interested in the behaviour of the system during its whole execution. The service compositions times should, as much as possible, have a small standard deviation of composition times. Figure 5.12 shows the system behaviour during its whole execution, using the reactive approach, when facing 20% failures. In a system using the *Reactive* approach, having 16k services and 20 % failures, the standard deviation of the service compositions times was large, what can be seen by the presence of many outliers.

On the other hand, Figure 5.13, shows the behaviour of the system, having 16k services and 20% failures, using the Delegate MAS approach which has a smaller standard deviation for the composition times, indicated by the presence of less outliers. We believe the difference in behaviour stems from the "memory" properties brought by the pheromones that each `ExplorationAnt` deposits in the environment, helping to guide other agents to use good component services.

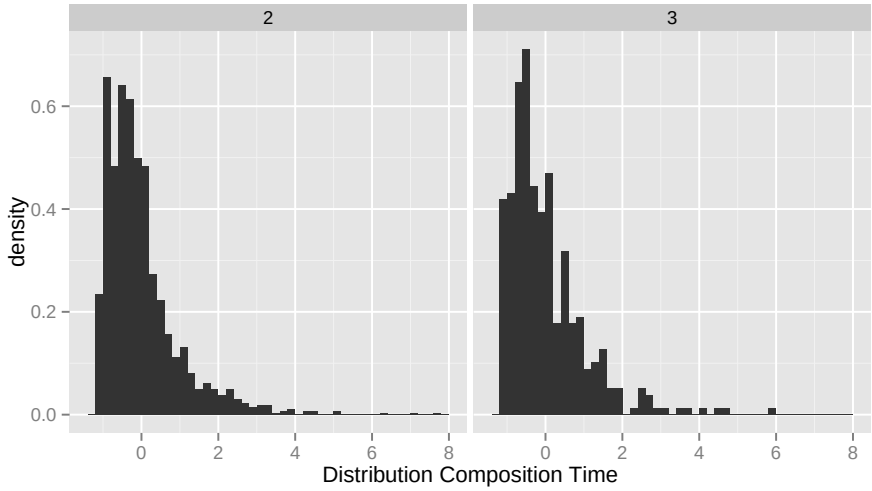


Figure 5.10: Normalized mean composition time using a Reactive algorithm, 16k nodes, 20% failures.

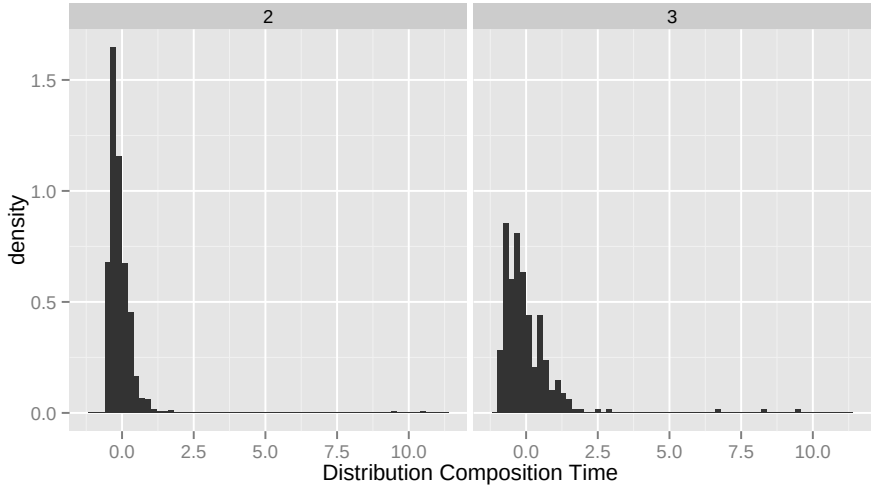


Figure 5.11: Normalized mean composition using Delegate MAS, 16k nodes, 20% failures.

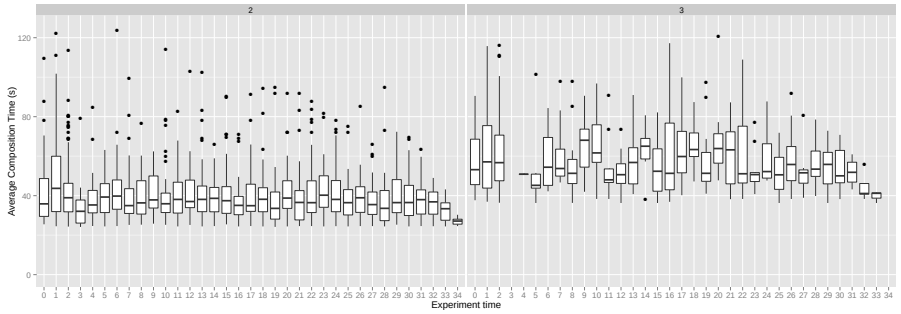


Figure 5.12: Reactive approach. Average composition times over the execution of the system with 16k services, under a 20% failure. The box-plots show a large standard deviation for the compositions, over time, what can be seen by the large number of outliers.

A shortcoming of our approach is that it produces bad service compositions (large composition time) when the system still does not possess enough information about good component services, which can be seen at the left hand side of Figure 5.13.

From the collected data, we can conclude that Delegate MAS provides superior performance and is less affected by small failures (20%) than a purely reactive approach. Delegate MAS works particularly well on moderately large service networks (16k services), and is slightly better than the *Reactive* approach on a very large network (64k services).

When the system is hit with large scale failures, that is, more than 80% of the services stop working, most part of composite services can not find suitable component services anymore, timing-out. However, the compositions which still manage to find suitable component services are not really affected by the failures.

Figure 5.14 shows a comparison of the averages of the composition times created by the Delegate MAS and *Reactive* approaches, in networks having 16k and 64k services and 80% failure rate. In a network having 16k services, Delegate MAS is 3.9% better than the *Reactive* approach when the composite service has only 2 tasks. Not like in the 20% failure rate scenario, Delegate MAS is only 8.1% better than the *Reactive* approach when the composite service has 3 tasks. In networks having 64k services, the reactive approach for compositions having 2 tasks was 41% slower than Delegate MAS. Again, in a network having 64k services, Delegate MAS performed better for compositions having 3 tasks.

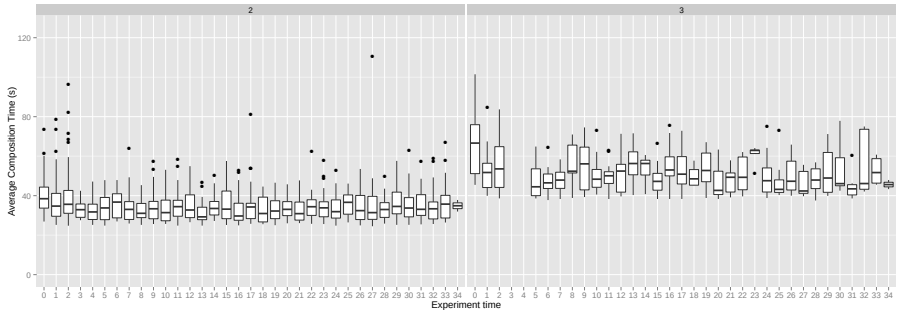


Figure 5.13: Delegate MAS approach. Average composition times over time, during the execution of the system with 16k services, under a 20% failure. The box-plots show that over the duration the experiment, Delegate MAS managed to maintain a small standard deviation for the composition times.

In compositions with 3 tasks, there was no significant difference between the approaches. An interesting result was that Delegate MAS 1.9% slower than the *Reactive* approach in a very large network with 80% failures.

A interesting aspect of very large scale failures in our experiments is that Composite services which manage to find suitable component services can still suffer a great variation in the quality of their compositions, what happens with both Delegate MAS and *Reactive* approaches. Large scale failures are very difficult to cope with, even our approach which performed particularly well on a system having 20% failure rate, becomes unstable when the system has a 80% failures rate. Figure 5.15 shows an execution of the system with 64k services, and 80% failure rate, using the Delegate MAS approach. We can see that many times no compositions are executed. That is due to the large number of component services unavailable at any moment.

5.5.3 Conclusion Hypothesis 3

In networks having 16k services and with a 20% failure rate, the *Reactive* approach selected component services for its compositions having 2 or 3 tasks which were, on average, 4.55% and 25.15% slower than our approach. We are aware that Delegate MAS performed better in the tested scenarios, but that it also has other drawbacks, such as communication costs.

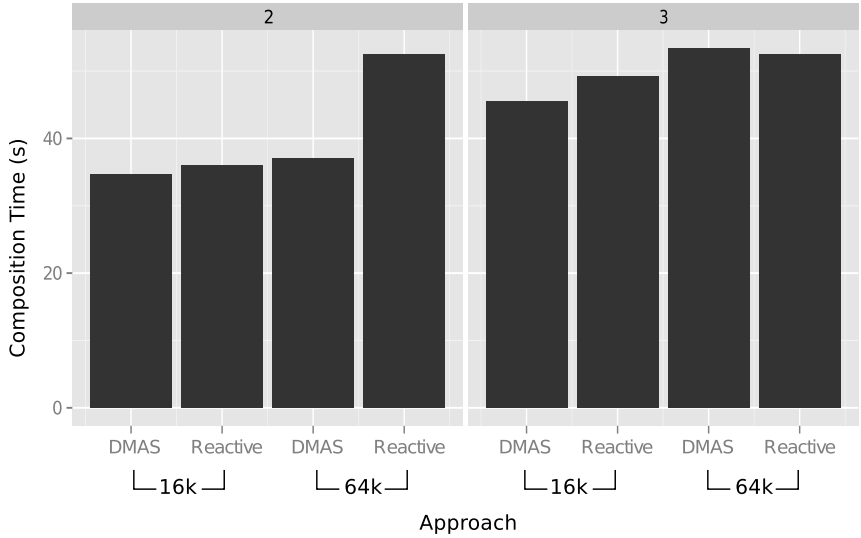


Figure 5.14: Comparison of the average composition times of service compositions created using Delegate MAS or the Reactive approach, in a system with 80% failure rate. As in the experiment with 20% failures, service compositions are made of 2 or 3 tasks. The average composition time difference between the two approaches is not so accentuated in this scenario, but Delegate MAS still manages to create the best compositions.

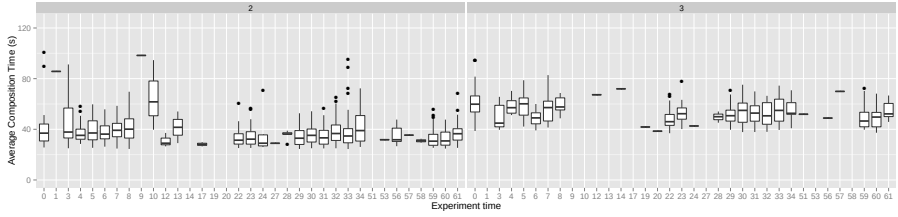


Figure 5.15: Overview of the composition times, when the system has an 80% failure rate. The system uses Delegate MAS approach to find,select, and compose from a pool of 64k services. The box-plots show a large standard deviation for the compositions, over time, what can be seen by the large number of outliers.

Chapter 6

Infrastructure and Validation

In the course of our research we studied the properties of Delegate MAS by creating simulations and prototypes of distributed systems. We learned that there are challenges to overcome to bring Delegate MAS and MAS in general, to the real world.

Such challenges are, for instance, how to integrate the agents from Delegate MAS into a programming paradigm of a service system. Another challenge is how can middleware support coordination in a decentralized system.

In this chapter we show two papers which illustrate: i) how to reify Delegate MAS concepts using standard WS-* technologies and middleware, and ii) a middleware providing reusable coordination abstractions. Such works were published in different venues, on which we discussed our research.

Service composition using standard WS-* tools

The first work we present in this chapter, entitled "MAS Organisations to Adapt your Composite Service" [30], is about the adaptation of service compositions to change in the QoS parameters of component services, using standard WS-* tools.

Our goal with this paper was to show the feasibility of using standard WS-* technologies with our MAS models and to perform runtime adaptation of composite services.

In this paper we study how to use MAS concepts to enable the runtime adaptation of composite services written using standard web-service technologies. We created composite services, using the BPEL language and deployed them on a standard BPEL engine. The BPEL engine was responsible to execute the many tasks of the composite service and to invoke the needed external component services.

Considering the goal of using standard WS-* tools, we connected a BPEL engine to an enterprise service bus, ESB, responsible for doing the actual invocation of the external services. That way, our MAS model could easily intercept the communication between the BPEL engine and the external services, and dynamically change which component service should be used for each service invocation.

A difficulty of adapting composite services is selecting which are good, regarding QoS criteria, component services to bind to. The MAS model we used, called Macodo [116], to represent possible component services and their quality attributes was responsible for maintaining the information about QoS parameters and deciding which service should be invoked.

The main contribution of this paper is to propose a high level MAS model that enables the adaptation of service compositions satisfying global constraints for the composition.

This paper shows: i) a MAS model used to adapt composite services to changes in the quality of component services; ii) a mapping between our model and WS-* concepts; iii) a implementation of the model using standard standard WS-* tools, showing the feasibility of this approach.

A middleware providing reusable coordination

Creating a coordination mechanism is error prone, due to different factors, such as the complexity of agent communication protocols, the asynchronous nature of the communication, and the interaction between the protocols and the internal behaviour of the agents. Because of this complexity we aimed at a coordination mechanism in a reusable fashion via a coordination middleware. We present, in Section 6.2, a paper entitled “CooS: coordination support for mobile collaborative applications” [29].

In this work we created a middleware focused on the creation of applications which need to use some form of coordination between their users, or resources. A coordination mechanism requires the exchange of a number of messages between the coordination participants, or partners. On the one hand, a coordination

mechanism may have strict constraints on the communication between the coordination partners, such as is the case of the `ContractNet` protocol. On the other hand, other coordination mechanisms may be less strict regarding the message exchanges of the coordination partners, such as is the case of Delegate MAS.

As is needed in a service selection and composition mechanism, the CooS middleware provides abstractions to handle the dynamic selection of partners. The contribution of this paper is to provide a middleware which abstracts the coordination mechanism complexities from applications which need to use coordination.

6.1 An Enterprise Service Bus Approach to Adapt Composite Services

This section presents our paper entitled "MAS organisations to adapt your composite service", presented at `MONA+ 2010` and published on the **Proceedings of the 3rd International Workshop on Monitoring, Adaptation and Beyond**, on the year of 2010, pages 33-39. The purpose of the research presented on this paper is to explore the use of standardized web-services technologies to create dynamic composite services.

Another purpose was to experiment if merging different agent coordination techniques, such as Delegate MAS and agents organizational structuring, such as the `Macodo` organizational model.

It is relevant to note that there are alternative technologies other than BPEL, to implement our coordination methods for service composition. A particularly robust technology is provided by the `Erlang` programming language [1]. `Erlang` is a programming language that can be used to create large scale systems which require high availability and robustness. Possibly, one would use Scalable Distributed Erlang to work with thousands of distributed services as we do [19]. Erlang also provides a HTTP libraries that can be used to connect to REST services for instance. Having this in mind, the paper starts below:

"MAS organisations to adapt your composite service"

The globalized world of business has created new demands for the architecture of distributed applications. These demands were shifted again with the creation of globally distributed supply chains [81]. The service-oriented

[1] http://www.erlang.org/

computing paradigm provides concepts satisfying the demands in this distributed environment [77].

Nowadays, complex business processes are modelled as composite services using BPEL. However, BPEL is not suited to work in very dynamic environments, leading to research on how to adapt processes written in this language. We focus on the problem of adapting a composite service in order to deal with global constraints, such as the End-to-End QoS, and the problem of preventing SLA violations.

We show how we can adapt an executing BPEL process in order to avoid SLA violations using the CASAS framework. We contribute to the state-of-the-art of composite services adaptation with our agent-based model.

6.1.1 Scenario

To illustrate the core ideas of this paper, we use a simple scenario from the Supply Chain Management domain based on interviews with the industrial partners of the DiCoMAS project[2].

A 4PL takes care of its clients' logistic procedures such as the transportation of materials between the client's factories. A 4PL has contracts with a number of carriers, called 3PL, that do the actual transportation. The 4PL's goal is to save time and money for its clients, by optimising transportation and business processes.

Each time our example 4PL receives a transportation request, it creates a transportation plan using an Automated Planning System (APS). The transportation plan is composed of a number of activities, each activity representing a transportation that should be made between two locations. The planning system splits the original transportation request in a number of sub-transports, because, normally, 3PL's are specialised in specific regions. Finally the transportation plan is written as a BPEL process and, after a first selection of 3PL's, deployed in a BPEL engine.

To execute this process, the 4PL's BPEL engine invokes the selected 3PL's web-services, informing them about the constraints, such as time constraints, monitoring constraints, etc.

How can we meet the quality requirements of executing a BPEL process within a specific time frame, even in the presence of partner failures, is the problem that we want to solve.

[2]DiCoMAS was a research project funded by the IWT (Agentschap voor Innovatie door Wetenschap en Technologie) research funding agency, Belgium

This problem is illustrated by our example `4PL`, that needs to do the transportation within a limited time period, as specified in the plan, but sometimes a partner that previously committed to do a transportation has problems and is not able to execute its part in the plan. For example, a small carrier, that has just one truck, is assigned a sub-transportation, but, suddenly, has a broken truck that needs to be repaired. In this situation, the `4PL` needs to find another carrier capable of doing the transportation for that specific sub-transportation, preferably within the same quality constraints.

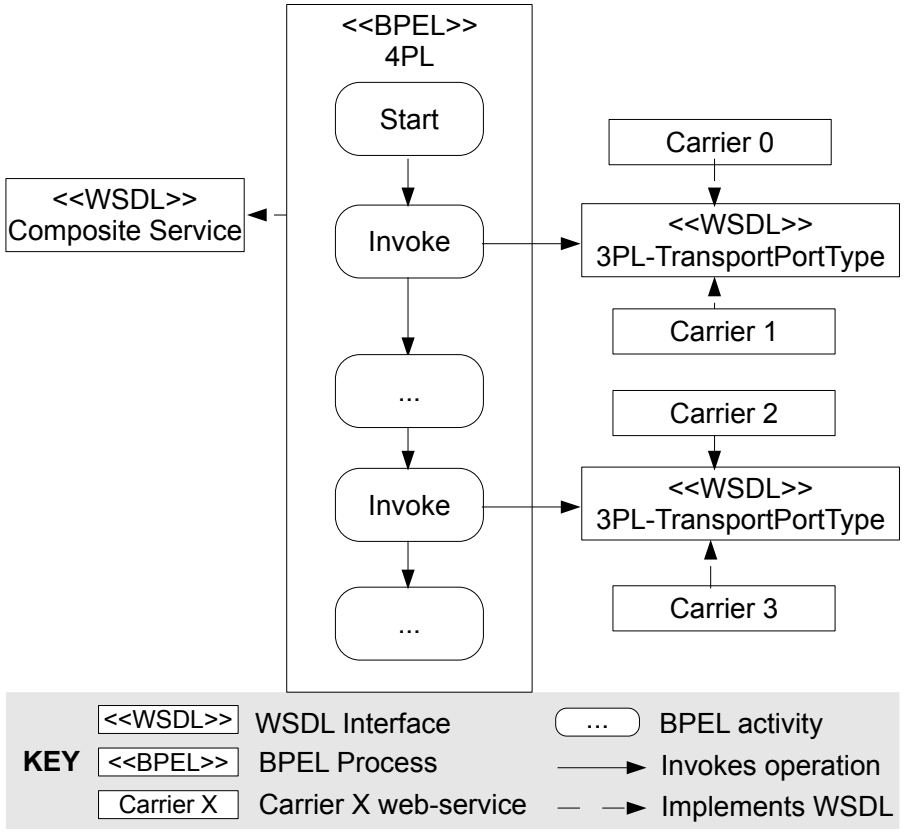


Figure 6.1: Transportation plan deployed in a BPEL engine.

Figure 6.1 depicts a simplified transportation plan. The transportation plan is deployed on a BPEL engine and is seen as an internal service within the `4PL`, simply called `Composite Service` here. Each `Invoke` activity in the transportation plan is executed by an external service, through the `threepl-TransportPortType` interface, which is implemented by partner

companies web-services. Each `3PL` must comply to the specified QoS, in this case the time to do the transportation.

In the next section, we show the concepts used in our solution to the problem of dynamic adaptation of composite services.

6.1.2 Another Level of Abstraction to Deal with Adaptations

The *de facto* standard for service composition is the BPEL language. BPEL can deal with primitive forms of adaptation, using `Dynamic Partner Links` and `Endpoint References` [48]. However, it is hard, if not impossible, to model the adaptations and its constraints required by a composite service using only BPEL.

We add another level of abstraction to the composite service model to solve the adaptation problem. This layer, called the organisation layer, explicitly represents the interactions between all the services participating in the composition, the adaptation constraints and the expected behaviour of the composition.

The Macodo Organisation Model

The MAS research community has a body of knowledge on Organisational Models. In this community, there are two distinct visions regarding these models: a) organisation being a first class entity, with its properties, states, laws [116]; b) organisation mainly as a process, composed by a set of steps to be taken by different actors [34].

MAS Organisational Models cope with collaboration between autonomous entities, called agents, working and interacting together, cooperatively or not, to achieve an organisation goal [31]. In our work we rely on the Macodo Organisation Model (Macodo), which defines the organisation as a first class entity with its own dynamics and separated from the participating agents.

The Macodo Organisation Model copes with context-driven dynamic organisations. It allows us to model complex collaborations between different entities, the agents, and to specify the rules that will trigger actions to adapt these collaborations [116].

Figure 6.2 depicts the domain model of the Macodo Organisation Model. The main concepts of the model are: `Organisation`, which contains roles and role positions; `OrgContext` which keeps the context information needed by the

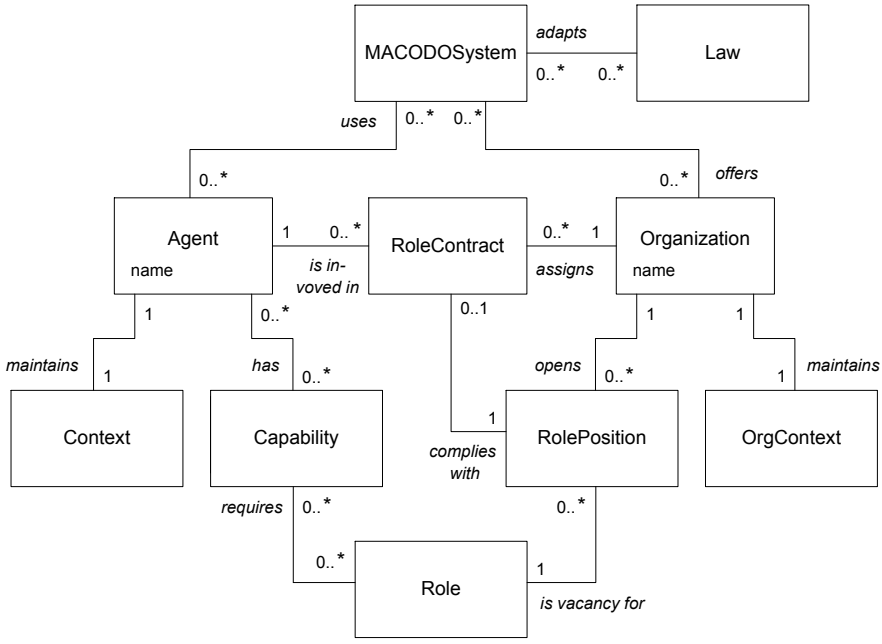


Figure 6.2: Domain Model of the Macodo Context-Driven Organisational Model [116].

organisation; `Role`, which constrain the behaviour expected from the agents; and `Context`, which is the contextual information needed by the organisation.

When joining an `Organisation`, an `Agent` takes a `RolePosition` in order to play a `Role`: this is represented by a `RoleContract`.

An `Organisation` encapsulates organisation rules used to adapt the collaboration. It actively inspects its context (`OrgContext`), *i.e.* the set of all the participating agent contexts, to enforce the organisation rules that need to deal with global constraints. A `Role` constrains the agent behaviour towards the organisation, requiring the agent to provide a set of capabilities and context. Using the `Roles` and agents' `Context`, the `Organisation` allows or denies the participation of the `Agents` in the collaboration.

There are two possible ways for an organisation to adapt: a well balanced combination of the two is the key to an adaptive organisation.

The organisation can access the context information of the agents and check

if the rules are being satisfied. That way, the adaptation is triggered by the organisation that keeps monitoring the collaboration between the agents. If one rule is not satisfied, it can change the state of a role and open a new role position, so that another agent can try to play that role in the organisation.

Another way for an organisation to adapt is through the agents behaviour. The agents can have a pro-active behaviour and monitor their own state. They can actively decide to leave or join an organisation. When an agent leaves an organisation, the organisation changes the state of the played role, opening a new role position, leading to its adaptation.

Mapping Macodo Organisations to Composite Services in BPEL

A BPEL process consists of: a) the BPEL code, which defines the execution flow; b) WSDL interfaces for the different consumed services; c) WSDL interface for the provided service (the composite service itself). A composite service specified in BPEL is made of a set of activities that are executed in a specific order. One special type of activity is declared using the `Invoke` construct, which invokes an operation in a partner link web-service.

In BPEL the communication with other web-services is done through the `PartnerLinks`, which define the relation between the BPEL process and partner web-services. Partner web-services are referenced by their `Port Type`, which is a set of abstract operations defined in WSDL.

We established a mapping between Web-services and Macodo Organisation concepts in order to create the organisation layer. This mapping is illustrated in Table 6.1.

Table 6.1: Mapping between Web-services and `Macodo` concepts

Web-services	**Macodo**
BPEL process	Organisation
PortType	Role
PartnerLink	Role Position
SLA	Capability
SLO	Agent Context

One BPEL process corresponds to one analogous `Macodo Organisation`. For each `Partner Link` specified in the BPEL process, we have a `Role Position` in the organisation. We have a `Role` for each `Port Type` in the BPEL process. The SLA are specified in terms of required `Capabilities` and, finally, SLO are specified as `Agent Context`.

We assume that the SLA and SLO are well understood and specified to the point that no human intervention is needed anymore. Otherwise it would be impossible to let the system adapt itself during runtime.

The CASAS framework uses this mapping and the BPEL process description to define the structure, such as the number of `Roles` and `Role Positions` of an organisation. Each BPEL process instance provides the correct flow of activities needed by the composite service and can be seen as the organisation functional behaviour. The agents operate the runtime binding to the real service providers and act as their representatives in the system.

Based on this mapping, separately from the BPEL process definition, the organisation provides a way to define adaptation rules that deal with the runtime adaptation that can occur during the process execution. For that, the agents provide context information to the organisation. But also, agents contain monitoring mechanisms and have pro-active behaviour to trigger adaptations (*e.g.* an agent can monitor its SLO and predict that an SLA will be broken).

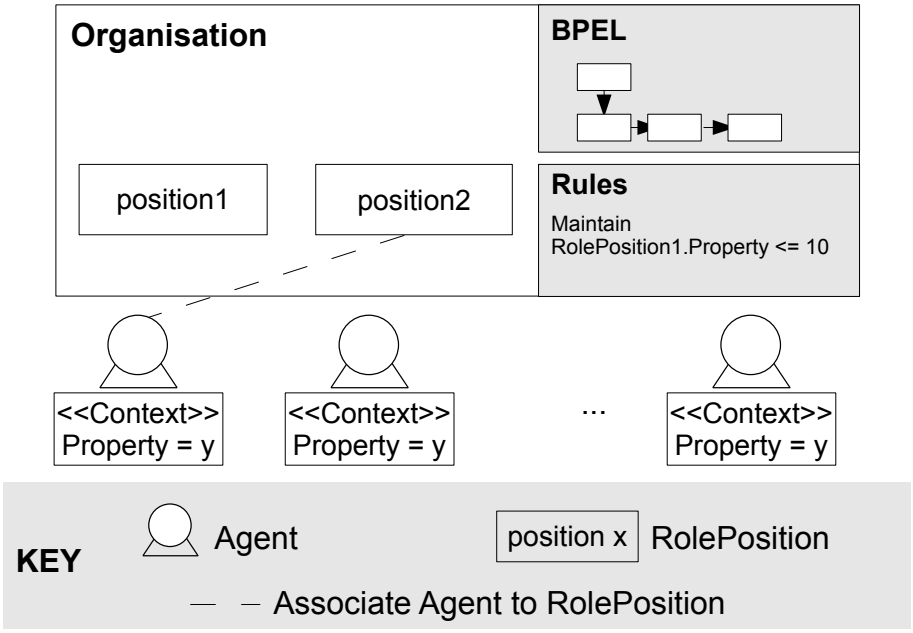


Figure 6.3: Conceptual solution integrating Macodo organisations, BPEL, and agents.

Figure 6.3 shows all the entities that collaborate in our system to create an adaptive composite service. It shows a BPEL process, the adaptation

rules, the `Macodo Organisation`, and the `Agents` that can participate in the organisation. The left most `Agent` is taking the `position2 RolePosition`, the other `RolePosition`, called `position1`, is not taken by any `Agent`. An `Agent` taking a `RolePosition` means that the agent is playing a specific `Role` in one `Macodo Organisation`. When the BPEL engine invokes an operation in a `PartnerLink`, the CASAS framework intercepts and redirects that invocation to the right `Agent`, which will, in turn invoke the operation on the actual web-service.

In the following section we introduce the CASAS framework, then we detail a prototype that gives example of adaptation rules and agent behaviours.

6.1.3 Casas Architecture

To handle the service-oriented concerns, CASAS rests on the Apache ServiceMix ESB. In particular, this ESB provides the Apache Ode BPEL engine to execute the workflows and the Apache CXF component to access external web services. A `NMR` is also present and is used by CASAS to intercept messages exchanged in the ESB and redirect them to the correct recipients.

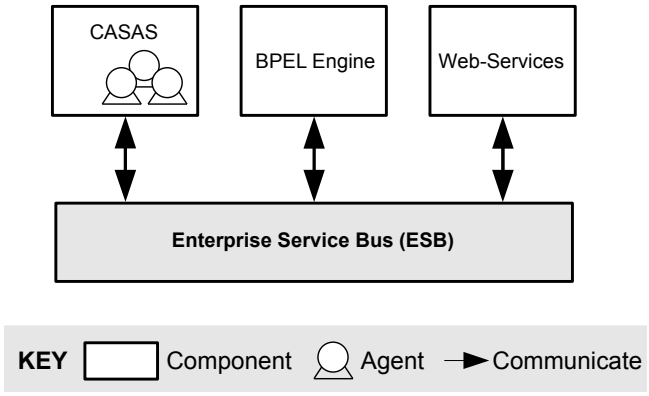


Figure 6.4: CASAS Framework Architecture.

In Fig. 6.4, depicting the high-level architecture of CASAS, we can see 3 important elements:

a) The `Macodo` System, which is responsible for managing the organisations (one per workflow instance). Each created organisation contains the rules that have to be enforced (the adaptation rules), a set of role positions to maintain

(the partner links) and the functional behaviour for the organisation (the workflow). The `Macodo` System enforces the rules in the organisation by opening and closing role positions when needed and also allows agents with a position in the organisation to play their role.

b) The set of agents in the system, which are responsible for joining organisations if they can take open role positions. They play their role when asked by the organisation and maintain a context that can be consulted by the organisation when required. Agents are parametrised by the web-service they represent, a social behaviour to decide when to join or leave an organisation and a monitoring behaviour to update their current context as well as trigger adaptations.

c) The CASAS system: each time a new workflow instance is launched, it creates a new organisation. It also creates the agents that will represent the different available partner web-services. The CASAS system uses the `NMR API` to set up the connection between: (i) each organisation and a particular instance of the workflow; (ii) each agent and represented web-service. The CASAS systems does this by listening and modifying exchanged messages.

From a service-oriented point of view, the CASAS system is responsible for dynamically providing the best partner services to the workflow instances. This selection is done in compliance to the SLA of each partner web-service. The BPEL engine doesn't know about the changes that can happen to the partner web-services, since it communicates with endpoints provided by the CASAS system. The CASAS system makes the connection between the BPEL engine and the real partner web-services, acting as a type of evolved proxy.

The MAS, explained in the next section, is transparent to the BPEL process and to the partner web-services, being used just internally by the CASAS system. From an agent-oriented point of view, the MAS is composed by the agents and their environment, *i.e.* the organisation that regulates their interactions.

6.1.4 Multi-Agent System

We used `SpEArAF` (Species to Engineer Architectures for Agent Frameworks) a development process presented in [72], to design the architecture and implement the MAS used in our prototype.

`SpEArAF` completes methodologies that mainly focus on the design of functionality, by promoting the engineering of application-specific frameworks for the development of multi-agent applications. By defining dedicated frameworks, the idea is to provide specific types of agents that fit functional requirements: developers can rely on the framework both when designing and implementing the MAS, thus forgetting operational concerns, and focus on the functional

behaviours of the agents. For example here, our objective is to make a framework for agents that are part of the type of Macodo organisation described previously and that interact with webservices through the ESB.

Frameworks are realised by assembling software components (possibly by reuse) in architectures for agents. Then, when programming the MAS, *hotspots* in the frameworks can be instantiated (possibly with sub-architectures) by the framework user to specify the behaviour of the agents using a set of agent-oriented and application-specific programming primitives defined by the framework. In practice, the architectures are defined using the MAKE AGENTS YOURSELF[3] tool that supports `SpEArAF` and the frameworks are implemented with `Java`.

Agent Architecture

Figure 6.5 depicts the architecture of one agent and its bindings to the Macodo System. As many agents as needed can be created at runtime and will have this architecture and bindings. An agent interacts with the system through a set of interfaces: he can receive *system events* about opened and closed role positions, based on that information he can *start a contract* with an organisation (*i.e.* take a position) and, when it has a contract for a role position, the system can send him requests to *play* his role and consult his *context*.

In the agent component itself we can see: a) a set of operational components (the frozen-spots of the framework) that implements the agent dynamics (`Message Dispatchers` to serially react to events) and the domain-specific mechanisms (`WebService Manager`) he uses; b) a set of behavioural components (the hotspots of the framework) that must be implemented to give a behaviour to the agent: handling organisation (`Social Behaviour`) events and playing the role (`Monitoring Behaviour`)

In terms of agent dynamics, agents are handling two types of events in parallel and reacting directly to them. Events from the system are handled serially by the `Social Behaviour` component, which then decides if the agent should take a position, eventually leaving another one. Play events are handled serially by the `WebService Manager` component that forwards the invocation to the real service represented by the agent. The `Context` of the agent is managed by the `Monitoring Behaviour` component. This component extracts information from the exchanged messages with the real service, which are, in turn, provided by the `WebService Manager` component.

[3] `http://www.irit.fr/MAY`

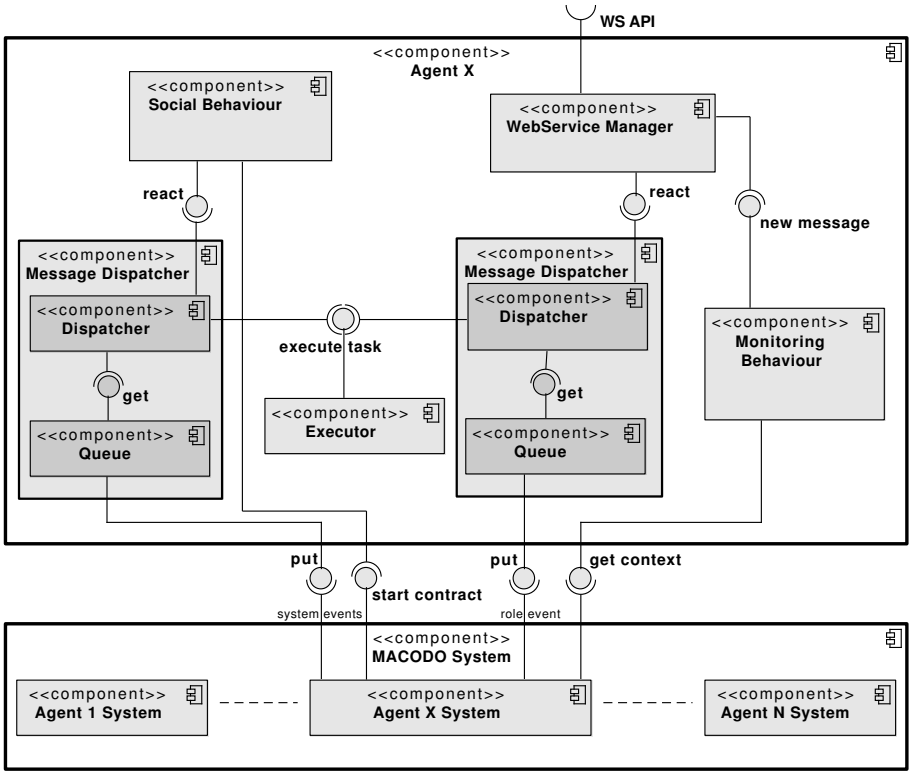


Figure 6.5: Agent Architecture.

The presented application-specific framework provides an agent architecture that explicitly exposes, through its components, the concepts available at the design level: social behaviour and monitoring behaviour. These behaviours can be implemented by the application developer (framework user) using the application-specific primitives without worrying about implementation of operational concerns. For example, in our case to implement the `Social Behaviour`, developers can use the `start contract` primitives using information provided in `system events`.

This architecture facilitates the development of the whole application by enabling clear separation of concerns and explicitly manipulating concepts from the organisation model, starting from the design down to the implementation.

6.1.5 Prototype

The prototype was written in `Java`, according to the described architecture. The software requires the BPEL process definition, the adaptation rules, the partner web-services addresses, the agents bounded to these services, and the agent behaviours.

Adaptation Rules

Adaptation rules are defined per role position and are parametrised by a function responsible for checking that the QoS indicators of the agents (its context) satisfies the defined SLA for the partner (a threshold). The organisation dynamics is hard-coded: it continuously checks the context threshold, it accepts, refuses and revises `RoleContracts` based on this context information.

Behaviours of the Agents

As said before, there are two parts of the agent behaviour: (i) social behaviour for joining and leaving the organisation; (ii) monitoring behaviour for managing the context. We implemented them as follow in a reactive way.

An agent that receives an event for an open position will try to start a contract if he has the capability of playing this role, *i.e.* he has a matching `PortType` and required SLA. If the agent decides to take a role position but the position is already taken, or if the organisation refuses to give the position to him, or if the position is closed while playing it, then the agent waits for new open positions.

The monitoring behaviour is strongly dependent to the type of context required by the organisation. This context is determined depending on the adaptation rules presented before. We used the delivery time of the `3PL` as the context in the studied scenario. The `3PL` provides this information through its web-service operations.

Runtime

When there is a request to start a workflow, CASAS instantiates the agents, with the configured behaviour, and the organisation, with its adaptation rules. Role positions are opened and agents join them. The organisation accepts the first agent that has the needed capabilities and provides the required context to take the role position.

The agents always update their context based on the delivery time reported by the 3PL's. If one agent does not comply with the organisation rules anymore, the organisation closes its role position and opens a new one. The organisation uses the agent's context and its own context to enforce a global constraint, which in our scenario is the total time to execute the transportation.

6.1.6 Conclusion

The main contribution of this work is to provide a high-level model which encompasses the composite service, partner services and the adaptation rules needed by the composition. This high-level model is provided by our framework abstractions, such as Organisations, Roles, Role Positions, Capabilities and
Agent Context, opening new possibilities for the adaptation of composite services, specifically in terms of satisfying global constraints for the composition, which can be specified in terms of the organisation context (OrgContext).

Another contribution of our work is that it allows to pro-actively adapt the service compositions. Instead of simply reacting to events and adapting the composition afterwards, agents can have pro-active behaviour and ask to leave the composition, before problems happen.

Service adaptation is a concern from the research community due to the dynamic nature of many problem domains and due to still not having a standard solution for adaptation. We borrowed ideas from the MAS research community, which has experience dealing with problems that demand adaptation. We presented the CASAS framework which uses the concepts from the Macodo Organisation Model and from Composite web-services, showing the mapping between two research domains concepts in order to deal with the adaptation problem.

6.2 A Coordination Middleware

Reuse is an important aspect of adapting composite services. If the techniques used to adapt the composition are too complex, the chance of designers using them are much lower. We explore how to reuse a coordination mechanism via a coordination middleware.

This section presents our paper entitled "Coos: coordination support for mobile collaborative applications", which was presented at Mobiquitous 2012 and published on the Lecture Nodes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, book Mobile

and Ubiquitous Systems: Computing, Networking, and Services, published by Springer Berlin Heidelberg, at pages 152 to 163.

In this paper we introduced the concept of a coordination middleware, its requirements and show a prototype of such middleware. The main idea behind our coordination middleware is to allow the reuse of the coordination algorithms, facilitating the creation of applications which benefit from such algorithms.

From the point of view of the application, the CooS middleware is accessed via an interface (the CooS Client Component), which provides two operations. One operation is to request a collaboration and the other operation is to register as a possible participant in a collaboration.

The main limitation of CooS is that it only provides extension points to coordination mechanisms based in auctions, such as the ContractNet protocol. This limits the types of application that can benefit from the middleware. Another limitation is that the middleware does not provide explicit extension points for adding new coordination protocol implementations. The only way to add a new coordination protocol is to create a new `Coos Coordination Component` from scratch.

We also do a study case on how to create an application to coordinate taxi drivers picking passengers on a city.

The paper starts below:

"CooS: coordination support for mobile collaborative applications"

The advent of mobile devices, such as smartphones and tablets, and their integration with cloud computing is turning ubiquitous computing into reality. This ubiquity opens doors to innovative applications, where mobile devices collaborate on behalf of their users. Applications that leverage this new paradigm, however, have yet to reach the market. One of the reasons is due to the inherent complexity of developing such collaborative applications on mobile devices.

In this paper, we present a middleware that enables coordination on mobile devices. middleware frees applications from directly managing the interaction between collaboration partners. It also uses contextual information, such as location, to dynamically determine possible collaboration partners. We focus on a particular class of applications in which mobile devices have to collaborate to allocate tasks (e.g., picking up passengers) to physically distributed resources (e.g., taxis). The technical feasibility of our middleware is shown by the implementation of our middleware architecture, a deployment of our middleware on a real cloud environment and operating it with over 800 clients.

6.2.1 Introduction

Every day, developers create dozens of new applications for smartphones and other mobile devices. Two important trends are making mobile devices the platform of the future. First, they provide a hardware platform, filled with technology, that is getting cheaper everyday. Modern devices come with communication technologies, like Bluetooth, GPRS, EDGE, and WiFi, and an abundance of sensors, such as accelerometers, compasses, altimeters, and GPS (Global Positioning System). Second, with the advent of cloud computing, it is possible to create applications that scale to serve hundreds of thousands of clients, while providing minimum delays, needed for near real-time mobile collaborative applications. In fact, cloud computing is changing the way computing is offered. Computing power has become a utility that applications can consume at will, facilitating the deployment of large scale mobile collaborative applications. Ubiquitous computing [112] is finally reality due to the advent of mobile devices and cloud computing, opening doors to innovative applications.

One particularly promising type of applications, are applications where mobile devices closely collaborate, on behalf of their users. These applications include crowd sourcing internet connections [75], collaborative traffic routing [11], collaborative scheduling of resources (e.g., cars) [85, 62], search and rescue systems [65], or allocating taxis to passengers in a dial-a-ride problem [59]. In the resource sharing problem, users can use the location information from their mobile devices to collaborate with other users in their vicinity, to organize on-the-spot car pools, for instance. Another very useful application is for improving public transportation with the use of autonomous vehicles, that could collaborate to find the best way to pick passengers [85].

Despite today's pervasiveness of mobile devices and the challenging problems that could be addressed using collaborations, applications that truly leverage the power of collaboration on mobile devices are still missing. One of the main reasons for this lack of applications is due to the inherent complexity of developing such applications. Mobile collaboration may also require users coordination. Existing coordination mechanisms, such as ContractNet [93], or MASCOT [87], require specific interaction flows involving large amounts of messages between coordination partners. Ensuring the correct implementation and execution of such mechanisms can be time consuming and error prone. Another problem is that collaboration partners are often not known in advance, but have to be determined dynamically, for example, based on their location. In addition, all these problems take place in a very dynamic environment, where everybody is moving, and where disconnections and changes in commitment are widespread.

To stimulate the future development of mobile collaborative applications, we need good middleware support that relieves developers of such complexities. In this paper, we present CooS[4], a middleware that operates providing common-middleware services [89] that enable the creation of decentralized collaboration of mobile devices. CooS targets a particular class of applications in which mobile devices have to collaborate to allocate tasks (e.g., picking up passengers), to physically distributed resources (e.g., taxis, autonomous cars). CooS addresses three key challenges:

1. dynamically determining collaboration partners (e.g., based on their location),
2. achieving scalable collaborations,
3. managing the interactions between collaboration partners.

The main contribution of this paper is a middleware to enable the creation of large-scale mobile collaborative applications. The novelty of our approach is to integrate location-based participant selection with coordination mechanisms, and offering this functionality as a reusable middleware service. The middleware service is designed to be deployed on any cloud computing provider.

Overview. The remainder of this paper is organized as follows: Section 6.2.2 describes the challenges faced to create mobile collaborative applications. Section 6.2.3 details the design goals and Section 6.2.4 the architecture of our middleware. We describe experiments of an application developed on top of CooS, and analyze their results, at Section 6.2.6. Finally, in Section 6.2.7 we present our conclusions.

6.2.2 Problem Statement

Our goal is to provide a middleware that supports the development of collaborative applications on mobile devices. Such applications typically require coordination of mobile devices to set up and execute the required collaborations. To illustrate the type of applications we want to address, we focus on the dial-a-ride problem for taxis. In this problem mobile devices have to collaborate to allocate tasks (i.e., picking up and dropping off passengers), to physically distributed resources (i.e., taxis).

The dial-a-ride problem has been extensively studied due to its applicability in various domains. The problem is computationally demanding, even for

[4]CooS: Coordination on Clouds.

small scale instances [78], and can involve various stakeholders with opposing goals. In the taxi problem, for example, taxi companies want to maximize their profit, typically at the expense of competing companies, and are even willing to compromise their quality of service (e.g., picking up a passenger on time). Passengers, however, want to be picked on time and reach their destination as soon as possible. The goal of the dial-a-ride problem is to pick up passengers in time, while maximizing the profit of all taxi companies.

Resources have a physical location and are mobile. Tasks are also location-based. Resources can commit to tasks (e.g., a taxi agreeing to pick up a passenger), de-commit to tasks (e.g., a taxi taking an alternate route), and can break down (e.g., a taxi breaking down). The number of involved resources and tasks can vary dynamically and scale up to thousands, for large collaborations.

In the rest of this section, we elaborate some key challenges in building mobile collaborative applications.

Key Challenges

Dynamically Determining Collaboration Partners based on Location. Classic coordination mechanisms, such as ContractNet or auctions, do not take location into account when determining possible collaboration partners. In our taxi problem, this would result in mobile devices of passengers interacting with the devices of all taxis in the system to find a possible resource. This leads to our first challenge.

> **Challenge 1.** A device should only collaborate with those devices whose location fits within the solution space of the underlying problem.

In our taxi problem, the mobile device of a passenger should only collaborate with the devices of taxis that are within a feasible range to pick up the passenger. Since both taxis and passengers are mobile, collaboration partners can change dynamically.

Scalable Collaboration. Each mobile device, active in the system, will have a communication overhead. This overhead can be related to the actual collaborations a device is involved in, but also to the process of finding the right collaboration partners. While a device may only have to collaborate with a few dozen of other devices, there can be thousands of devices that are all potential collaboration partners. Finding the relevant collaboration partners may induce a communication overhead that is disproportionate to the overhead induced by the actual collaboration. This defines our second challenge.

> **Challenge 2.** The communication overhead of a device in the system, related to finding relevant collaboration partners, should be independent of the total number of devices in the system.

The communication overhead of each device in the system is only dependent on the number of devices it directly collaborates with.

Managing the Interactions between Collaboration Parters.

Coordination mechanisms tend to get complex, requiring asynchronous interactions with complex message flows. Current technologies, such as GCMA (Google Cloud Messaging for Android) [5], only provide a basic messaging mechanism for the interaction of cloud services and mobile devices. Managing these interactions can be time consuming and error-prone. Reuse existing coordination mechanisms could greatly improve these problems. Achieving such reuse, however, requires a clean separation between application logic and coordination logic, which poses an even bigger problem. This leads to our final challenge.

> **Challenge 3.** Coordination mechanisms and their required interactions should be easy to manage, allowing developers to separate application logic from coordination logic, while promoting reuse of existing coordination mechanisms.

Requirements for the CooS Middleware

Given the challenges for developing mobile collaborative applications, we can derive a set of functional and non-functional requirements for the CooS middleware. There are two main functional requirements for the CooS middleware:

1. **Dynamic Partner Selection.** The middleware dynamically selects the relevant collaboration partners based on their location.
2. **Managing Interactions between Collaboration Partners.** The middleware enforces the coordination mechanisms, chosen by the application developer, ensuring the required interactions take place without violating message flows or timing constraints.

[5] http://developer.android.com/guide/google/gcm/

We can also derive two non-functional requirements for the CooS middleware:

1. **Scalable Partner Selection.** The middleware ensures that communication overhead, of each device, related to participant selection is independent of the number of devices in the system.
2. **Encapsulation of Coordination Mechanisms.** The middleware encapsulates the coordination mechanisms and related interactions as reusable middleware services. The middleware provides an API to application developers that allows to separate application logic from coordination logic.

6.2.3 Design of the CooS Middleware

Before explaining the CooS Middleware architecture in detail, we provide a high-level overview of its design and motivate the most important design decisions.

Providing coordination mechanisms as a reusable middleware service. A key requirement of the CooS middleware is to manage the interactions between collaboration participants, relieving application developers from the related complexities. To do so, the middleware provides a set of predefined coordination mechanisms as reusable middleware services. Applications can then choose the proper coordination mechanisms according to their needs.

Using an event-driven architecture to enforce coordination mechanisms.

To provide the coordination mechanisms, the middleware needs to enforce the required interactions between the collaboration partners. To do so, the CooS middleware relies on an event-driven architecture. Each coordination mechanism is defined as a set of interaction events (i.e., sending and receiving messages) that have to take place in a specific order and within particular timing constraints.

The event-driven architecture is particularly suited to handle the continuous internet connections and disconnections of mobile devices. It also allows to create a thin middleware layer to be deployed on mobile devices, which are typically computational constrained.

Using location-based Publish/Subscribe to select partners. Another key requirement of the CooS middleware is to dynamically select coordination partners based on their location. This avoids interaction with irrelevant participants, such as taxis in other cities. To achieve this dynamic partner selection, the CooS middleware employs a location-based Publish/Subscribe

mechanism [40]. The location-based Pub/Sub system allows to subscribe to events, based on the location or region in which an event occurs. Every time a new event is created in a location, the subscribers to that location or region receive a notification. Publishers of events attach location information to their events, so this information can be used to match interested subscribers.

Using the location-based Pub/Sub system, the middleware notifies the relevant applications whenever a new collaboration is triggered within their regions of interest. To do so, the CooS middleware maintains the location of each mobile device active in the system.

Offloading coordination-specific functionality to mobile devices.

Providing coordination mechanisms and dynamically selecting coordination partners requires functionality such as determining the location of mobile devices, or calculating the shortest path from a passenger to a taxi. The CooS middleware relies on the capabilities of modern devices to offload these tasks to the devices themselves. The CooS middleware uses the GPS of the device, for example, to determine the location of taxis or passengers, and the locally available routing software to calculate possible paths.

Using a cloud-based infrastructure.

While mobile-devices can be used to offload some of the coordination-specific functionality, the actual enforcement of coordination mechanisms and selection of collaboration partners can put a heavy burden on the mobile devices, if done locally. To relieve the mobile devices, the CooS middleware relies on a cloud-based infrastructure to enforce the coordination mechanisms and to determine the possible collaboration participants.

The cloud-based infrastructure also provides a more uniform communication channel. Many times, mobile devices cannot communicate directly with each other, because they are situated behind proxies or firewalls. The cloud, however, is able to provide a uniform messaging layer to all mobile devices.

To provide additional scalability to applications, middleware services deployed on the cloud can easily be replicated to more computers. This allows to scale-out applications to match the current number of users.

6.2.4 CooS Architecture

The runtime architecture of the CooS middleware consists of two main components: the CooS Client Component, deployed on each mobile device, and the CooS Middleware Component, deployed on a cloud provider.

The CooS Middleware Component has two main responsibilities: (1) dynamically selecting relevant collaboration partners based on their location, and (2) enforcing a particular coordination mechanism, chosen by the application developer, among the selected partners.

The CooS Client Component serves as a mediator between the CooS Middleware Component and the application. It provides an API that allows the application use the coordination mechanisms provided by the CooS Middleware Component. A user has two possible ways of collaborating with other users, as an initiator, triggering a collaboration, or as a participant, waiting for collaboration requests, the CooS Client Component allows applications to play two possible roles: the initiator role or the participant role. In the taxi application, for example, the application plays the initiator role at the passenger's device, and the participant role at the taxi driver's device.

We illustrate the middleware architecture with ContractNet as coordination mechanism (Sect. 6.2.5), and briefly discuss the implementation of the CooS middleware architecture (Sect. 6.2.5).

6.2.5 CooS Middleware Component

The CooS Middleware Component uses an event-driven architecture to enforce its coordination mechanisms, and relies on a location-based Publish/Subscribe mechanism [40] to dynamically select the collaboration participants. The internal architecture of the CooS Middleware Component consists of four components: a Coordination Component a Location-Based Publish/Subscribe Component, a Location Store, and an Event Dispatcher (Fig. 6.6).

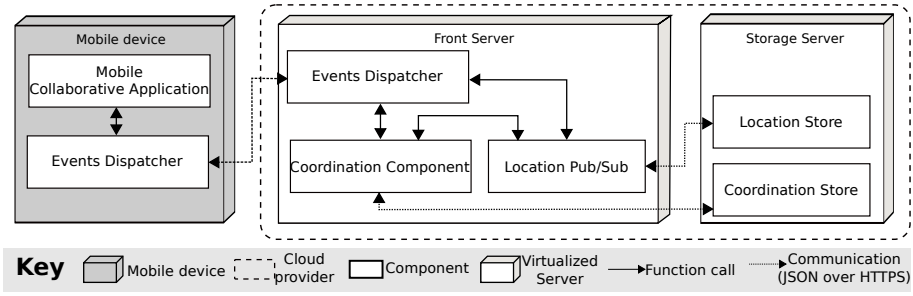


Figure 6.6: Deployment view of CooS middleware on a cloud provider.

The Coordination Component is responsible for enforcing the selected coordination mechanisms among the active participants. This includes making sure that interaction events (i.e., sending or receiving message) take place in the right order without violating any timing constraints. The

Coordination Component is also responsible for maintaining the state the ongoing coordinations. Coordination mechanisms can define constraints based on location information. The Coordination Component is a publisher and a subscriber of events from the Location-Based Publish/Subscribe Component.

The Location-Based Publish/Subscribe Component provides the functionality for location-based participant selection. Active devices publish their location information using the CooS Client Component. The location information is processed by the Location-Based Publish/Subscribe Component and persisted on secondary storage.

The Event Dispatcher is responsible for receiving events and dispatching events from and to the CooS Client Components. The Event Dispatcher relies on a unique *DeviceID* to identify each CooS Client Component, allowing to have asynchronous interactions between CooS Client Component and CooS Middleware Component. Interaction between the Event Dispatcher and the CooS Client Components is based on stateless protocols, such as HTTP.

CooS Client Component

The CooS Client Component acts as a mediator between the CooS Middleware Component and the application. It provides an asynchronous API to applications to use the coordination mechanisms provided by the CooS Middleware Component. The main API operations are illustrated below:

```
requestCollaboration(DeviceID device, Coordinates location, Payload
payload, InitiatorCallback cb)

registerAsParticipant(LocationCallback lcb, ParticipantCallback pcb)
```

To start a collaboration the application uses the *requestCollaboration* operation of the CooS Client Component. The CooS Client Component, in turn, creates an event including the *DeviceID*, the location of the device, and an application-specific payload. The CooS Client Component then dispatches this event to the CooS Middleware Component. When invoking the *requestCollaboration*, the application needs to pass an *InitiatorCallback*. This callback is specific to the coordination mechanism, and provides the actual functionality of the application to be the initiator of the coordination. For example, when using the ContractNet coordination mechanism, the callback should provide the functionality to inform the application with the outcome of the ContractNet protocol. The *Payload* is application specific data not inspected by the middleware. The middleware only passes this data back to the application.

To participate in collaborations, applications have to register two callbacks, using the *registerAsParticipant* operation of the CooS Client Component. The

first callback is the *LocationCallback*. This callback is responsible for providing the middleware with the proper location information, required by the location-based participant selection of the CooS Middleware Component. The second callback is the *ParticipantCallback*. Like the *InitiatorCallback*, this callback is specific to the coordination mechanism, and provides the actual functionality of the application to be a participant in the coordination.

Illustration of the CooS Middleware Architecture

To illustrate the CooS middleware architecture, we show how applications can register as participant and how applications can request collaborations. To register as participant, applications call the *registerAsParticipant* operation on the CooS Client Component (Fig. 6.7). The local CooS Client Component then starts a process that will retrieve the application-specific location on regular intervals from the application, and send location updates to the CooS Middleware Component. The CooS Middleware Component stores these locations in its location store. Once registered as a participant, the CooS middleware will take these applications into account when selecting the relevant collaboration partners for each new collaboration.

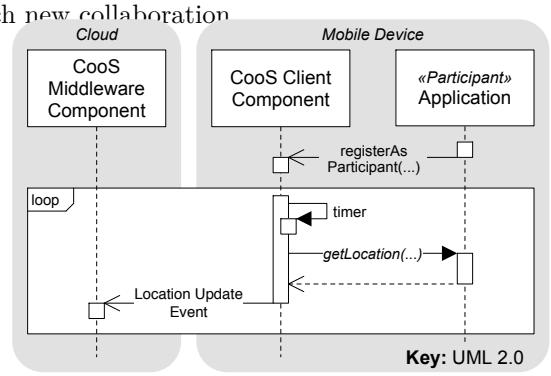


Figure 6.7: A sequence diagram showing how the CooS middleware maintains the location of each potential collaboration participant.

When an application starts a collaboration, it calls the *requestCollaboration* operation on the CooS Client Component (Fig. 6.8). The CooS Client Component sends this request to the CooS Middleware Component, which selects the relevant participants, among the registered applications, based on their stored location. The CooS Client Component of each selected participant is then informed about the collaboration request. These CooS Client Component's will then start a coordination-specific interaction with their local application (in Fig. 6.8, this interaction is shown as generic *collaborationCalls* and *Coordination Events*). All interaction events between participants pass through the CooS

Middleware Component, which uses the context of active coordination sessions to act as an interaction hub.

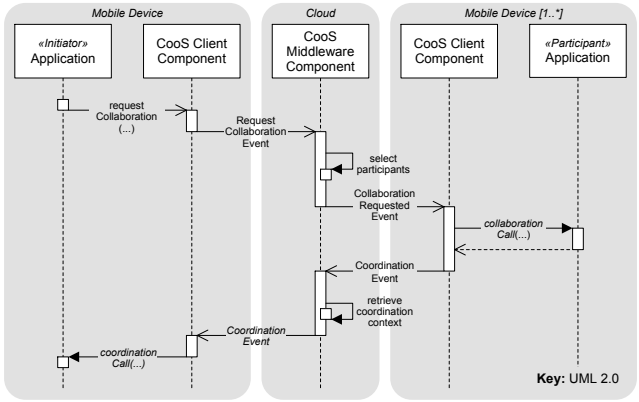


Figure 6.8: A sequence diagram showing how an application can initiate a collaboration.

Implementation

The CooS prototype uses off-the-shelf technologies. The CooS Middleware Component uses Node.js[6], a high-performance event-driven application server for networked applications. The CooS Middleware Component maintains location and on-going coordination information, which is stored on a mongoDB[7] database. MongoDB is a scalable, high-performance, open-source NoSQL database.

The CooS Client Component and CooS Middleware Component have bi-directional communication, so that the coordination interactions can happen, with the cloud notifying the mobile devices and vice-versa. The prototype communication is made using the WebSockets [43] protocol.

6.2.6 Evaluation

Case study: Using smartphones for coordinating taxis in Brussels

We performed a case study in order to evaluate the technical implications of using our middleware in a more realistic setting. We implemented a coordination

[6]http://nodejs.org/
[7]http://www.mongodb.org/

application to coordinate all the taxis in Brussels on their task of picking passenger and delivering them at the requested locations.

Our goal with this case study was to check the technical feasibility of using our middleware for such problem. Coordinating taxis consists in allocating the taxi that can pick a passenger in the shortest time, that way minimizing the passenger waiting time. Passengers have the application installed on their mobile phones. When a passenger wants a ride, he simply indicates when he will need a taxi and where he wants to go. This information, together with the location information given by the GPS of the passenger's mobile device, is sent to all taxis that are interested in picking passenger and delivering them in a particular region.

Evaluation system model

We have implemented a prototype version of our middleware, and deployed the **EventSignaling** part of our middleware on the Heroku [8] cloud provider.

We setup 80 computers to participate in the emulation, executing the taxi application. Every computer having 10 instances of the taxi application running as independent processes. Besides the taxi applications, we also setup 8 computers to simulate the passengers. Every computer executing 10 instances of the **PassengerApp**.

Hence, in our emulation we executed 880 instances of an application using our middleware. Each instance had a very simple simulator, responsible for issuing commands to the application. The commands consisted in simulating a taxi driver driving a taxi following a particular route and in passengers asking rides on their mobile phones.

We developed two simple components to simulate the behavior of a passenger and a taxi driver using our application. The simulators have the following behavior:

- **Passenger Simulator**, reads a location from the destinations list and asks a new ride to the *PassengerApp*. When the *PassengerApp* indicates the ride is done, the *Passenger Simulator* requests a new ride. Otherwise, if the *PassengerApp* indicates there is no taxi available, the *Passenger Simulator* chooses the following location from the destinations list and issues a new ride.

[8] http://www.heroku.com

- **Driver Simulator**, simulates a taxi moving into the location of a passenger. It does this by virtually following a route given by the *TaxiApp*.

On a real world deployment of our application it would be possible to configure the location updates issued by the middleware to one update every few seconds, or more. However in our emulation we configured the middleware to issue a location update every 100 ms. What in our experiments lead to 8800 requests per second without any delay due to the number of requests. The application showed delays when handling more than 14.000 requests per second. The operation of the middleware at the client side is negligible, while the operations at the CooS Middleware Componentheavily relies on the performance of the cloud provider. The main shortcoming can be the response time due to the internet connection of the mobile devices.

Regarding the implementation of the taxi application, we learned that using the GPS (Global Positioning System) of mobile devices has to be done carefully in order to avoid draining the device's battery. Another lesson we learned from implementing the taxi client application is that delegating the communication complexity to the CooS middleware facilitated the application development, however it was still complex to manage all the callback functions needed by CooS.

6.2.7 Conclusion

In this paper, we presented CooS, a middleware that enables the creation of collaborative applications on mobile devices. CooS targets applications in which mobile devices have to collaborate to allocate tasks (e.g., picking-up passengers) to distributed physical resources (e.g., taxis). CooS addresses several key challenges for developing mobile collaborative applications. These challenges include dynamically determining collaboration partners, achieving scalable collaboration, and managing the interactions between collaboration partners.

We presented a middleware architecture for CooS that encapsulates coordination mechanism as a reusable middleware service for applications. This encapsulation provides a clear separation of concerns, freeing application developers from handling coordination-specific complexities. The evaluation of CooS showed the technical feasibility and scalability of the presented middleware architecture. As future work, we plan to perform an empirical study, with real software developers, to assess how CooS impacts the development of mobile collaborative applications.

6.3 Lessons Learned

Designing, implementing and testing our coordination models was challenging. Creating the software to support composite service adaptation using standard web-services was extremely cumbersome. The most popular ESB at the time was still in development and had almost no documentation. Besides the lack of proper documentation, it was very difficult to deal with all the XML configurations needed to add new components to the ESB. The communication between the component implementing our model and the BPEL engine was done via message passing, what forced us to have a lot of plumbing code to adapt the messages.

We learned that research involving "real world" WS-* technology demands a great effort on engineering, and not so much on the conceptual aspects of the research. However, it was very instructive to learn how to deal with standardized technologies and to reach its limitations.

While creating our coordination middleware, we learned that using callbacks to handle asynchronous communication can lead to very complex implementations. Even for simple protocols, it is quite easy to arrive a layers of nested callbacks, which are extremely hard to test or debug. Having this complexity in mind, we learned that dealing with event-based systems using callbacks is not a good approach.

Because of the experience we gained while working on the papers presented in this chapter, we decided to change the approach we were using the create our coordination mechanisms. We started using a reactive approach, based on the actors model [1], which provides a much clearer way to handle the inherent complexity of our coordination mechanisms.

Chapter 7

Related Work

This chapter situates the work done at this thesis relating it to current state-of-the-art research on service compositions. There are different research fields which are related to our work. The research related to coordination techniques is covered in the Chapter 2, Section 2.3.2.

The first section of the related work highlights other domains where Delegate MAS was successfully used.

BPEL was the standard way to describe service compositions in many research works, however currently the adoption of BPEL is very limited. The second section of the related work present works which perform service selection and composition, and adaptation, relying on BPEL and in other languages.

Ideally coordination mechanisms would easily be reused, as middleware for instance. We also present works which offer reusable coordination abstractions to application developers.

We finish the related work by showing coordination applied to the domain of resource allocation in cloud environments.

7.1 Delegate MAS in other domains

Many Delegate MAS principles have been applied to the traffic domain [23, 22]. For example, the work of Claes et al. [23] applies the intention propagation, and exploration concepts of Delegate MAS to the vehicle traffic domain. Current vehicle guidance systems use real-time information to route traffic. The main

issue of such systems is that they may only react to traffic jams, but can not prevent them. Claes's approach creates a virtual model of the road infrastructure where agents navigate on behalf of the vehicles. These agents, which have an exploratory behaviour based in ACO, explore the environment looking for traffic forecasts. The agents are capable to reroute vehicles so that it is possible to avoid traffic jams.

Delegate MAS was also applied to the pickup-and-delivery (PDP) problem [74] and to manufacturing systems [108, 105].

The work of Verstraete et al. [108], for instance, discusses how to use two complementary approaches, named PROSA and Delegate MAS in engineering manufacturing control systems. Delegate MAS is mainly used to support the decision taking process, with agents deposing pheromones about the machinery they intend to use, allowing then optimizing the machinery usage of the factory.

7.2 Infrastructure Support for Adapting Composite Services

The most common form of service adaptation is through late-binding, also called vertical adaptation [76]. This type of adaptation doesn't change the structure of the composite service, in terms of the order of executed activities for instance. Instead, it changes the component services providing participating at the composition.

Many works focus on adapting composite services defined in BPEL and deployed in BPEL engines.

The work of Anja Strunk et al. [96] tackles the adaptation problem in controlled environments, where the alternative services are known at design time. The designer of the composite service indicates, at design time, which are the alternative services that can be part of the composition. The reasoning behind this approach is to leverage standard BPEL engines, instead of extending or adapting them.

The work proposes an architecture made of three main components: a BPEL engine, a monitoring component and a rebinding component. The BPEL engine is responsible for executing the BPEL process definition. The monitoring component watches events and triggers the rebinding component, when SLA violations are detected. Finally, the rebinding component replaces a failing service with an equivalent one, which was previously specified at design time.

The work of Braem et al. [14] provides tackles many limitations of the BPEL language, specially the lack of separation of concerns. The approach, called **Padus**, extends BPEL by adding aspect-oriented decomposition and composition mechanisms to the language. Padus allows the introduction of crosscutting behaviours to existing BPEL processes. That way, it is possible to control the complexity of core processes by improving the separation of concerns and freeing these core processes of activities which are not fundamentally part of them. Besides improving the BPEL language, Padus processes remain compatible with existing infrastructure.

`TRAP/BPEL` is a framework that enables the creation of adaptive service compositions by instrumenting BPEL processes [41]. This approach relies on binding the composite service to proxies. The proxies then perform the binding to the real component services.

The approach relies on automatically modifying the BPEL processes to monitor a set of component services. Every time there is a problem on a request, such as an invocation failure, or timeout for instance, the BPEL process redirects the request for service is redirect to a proxy service. The failed component service is then replaced by a properly functioning one.

Another approach to adapt BPEL processes is through the use of an ESB connected to a BPEL engine and to a service adaptation engine. Massimiliano Colombo *et al.* [26] applied the ideas of Autonomic Computing, such as self-configuration, self-healing, and context-awareness to create a platform called `SCENE`. This platform is used to tackle the problem of dynamically reconfiguring composite services.

`SCENE` provides a rule language that is interpreted on a Drools engine. The rules are checked at runtime and are used to realize the correct bindings between the BPEL engine and the concrete services. The rules are specified in terms of events that are generated by activities specified in the BPEL definition.

To adapt a running BPEL process, the platform intercepts all events and reroute them to the Rule engine. The rule engine checks what rules match the event and process it, possibly leading to the rebinding of a service.

The `SCENE` platform architecture implements the Message Router Enterprise Integration Pattern [52], with the rules defining the routing logic. `SCENE` is a very good solution and our architecture resembles it.

Composition and Adaptation Languages

Annapaola Marconi *et al.* [67] approach provides adaptation through the concept of adaptive pervasive flows `APF`. `APF` are specified using standard control elements, such as sequence, choice, and parallel operators. The flows are modeled in a way that they are logically attached to physical entities, and can be used to model workflows that are related to specific objects.

In this work, the researchers created a new language called APFL, which is an extension to BPEL. The new constructs take the context into account in order to adapt the execution of the business process, providing, for instance, more alternative flows than the standard BPEL constructs.

Anis Charfi *et al.* [18] proposes a plug-in architecture for self-adaptive web service compositions. In their approach, the orchestration engine is extended with adaptation plug-ins. Each plug-in has a well-defined objective and is developed by domain experts.

The plug-ins are written in the form of aspects, the engine is based on `BPEL4AOP`, defining pointcuts and advices. The plug-ins can be loaded at runtime and engine weaves the aspects specified in the plug-ins into the running BPEL process. This solution provides a mechanism to adapt any BPEL activity even before is has happened, because of the use of aspects. Our approach does not assume any particular BPEL engine, because we do not need to modify it to insert monitors or actuators. Instead, we use the ESB infrastructure to intercept messages sent by the BPEL engine, decoupling our solution from a particular BPEL engine implementation.

Philipp Leitner *et al.* [63] proposed a framework called `PREvent`, which is a system that integrates monitoring, prediction, and adaptation of service compositions. The main goal of the `PREvent` framework is to adapt service compositions in order to prevent SLA violations.

The framework mainly consists of three components: Composition Monitor, SLO Predictor, and the Composition Adaptor. The Composition Monitor is responsible for monitoring the runtime data. Prediction of violations are handled by the SLO Predictor, which uses learning techniques to identify the services that can cause SLA violations in the future. Finally the Composition Adaptor component is responsible for identifying and applying adaptation actions.

The PREvent framework is presented in [63]. PREvent is a service framework that encompasses, monitoring, adaptation and prevention of composite service failures. The framework tries to prevent composition failures by using learning algorithms that try to predict which component services will fail during the execution of the composition. When the prediction algorithms indicate a possible

failure the framework adapts the composition by selecting another component service. The main goal of PREvent is to guarantee the compliance the the composite service SLA. PREvent also improves the robustness of the composite service as a result of pursuing the SLA compliance.

From the field of MAS, tackling the adaptation of a composition of partners has been studied in [24] in the context of manufacturing control. In this work, in the same way that sequences of services must be provided by different partners in a composite service, containers have to be manufactured by different machines. The objective of the system is to be as efficient as possible while handling the dynamics of the system (new containers arriving, machine failures, priorities...). Machines, operators and containers are embodied as agents. Here, the composition as well as the adaptation is expressed only from the point of view of the agents by means of local interactions and results in the self-organization of the whole system.

There is plenty of research about selecting component services to participate in service compositions [6, 122, 69, 5, 66]. These works focus on creating algorithms capable of selecting the best available component services, in terms of QoS. Our work shares the idea that it is possible to improve properties from composite services by selection and biding to component services at runtime. However, in our research we study the problem of not only selecting the best available services, but also selecting component services that will lead to robust compositions.

The work presented in [46] discusses and evaluate different techniques for component service selection. The simulation results show that the most efficient approach is the proxy-based, followed by the collaborative approach. In the proxy-based approach, all the service invocations go through a proxy that can then, load balance and select the best available services. In the collaborative approach different composite services collaborate, sharing QoS information about component services, to allow a better component service selection. Our solution shares characteristics with both the proxy-based and collaborative approaches. We focus, however, on the creation of robust compositions instead of focusing only on minimizing a certain QoS metric, such as response time or price.

The work on [94] explicitly focus on creating robust service compositions. The work uses decision theory for dealing with the uncertainty associated with component service providers. It proposes a mechanism for component service selection that explicitly takes the reliability of the created composition into account. The service selection algorithm takes the most critical tasks into account and use service redundancy for these tasks. The algorithm also uses planning techniques to create contingency plans, in the case of component

services failures. Another interesting characteristic of the algorithms presented in [94] is the use of service reservation for parts of the composite service. Our work also takes the robustness of the composition into account when selecting component services. However our approaches differ in how to create robust compositions. Our approach relies on the aggregated information available in the agent overlay network and in advanced reservations of services that will participate in the composition.

7.3 Middleware Technologies

Our middleware does not deal with low level communication issues, instead it facilitates to coordinate the task allocation between several entities participating in an application. [89] proposes a layered view to position the different types of middleware available. Our work fits into the **Common Middleware Services** layer , as illustrated in Figure 7.1, , since our middleware provides a higher-level domain-independent component that allows application developers to concentrate on programming application logic, rather than focusing on low level hurdles specific to the coordination protocol in use.

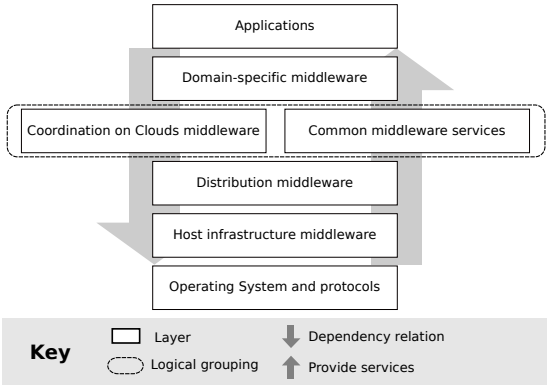


Figure 7.1: Contextualizing CooS relating to middleware layers.

The work [51] adds quality-of-service guarantees to middleware which works upon the elastic resources from cloud computing. It shows a technique to guarantee a specified quality-of-service even on a changing cloud environment.

There are several works exploiting middleware as a way mitigate different challenges associated with the application development for mobile devices [47],[88], [104], to cite a few.

The work [88] provides a number of abstractions to deal with mobile applications. The main goal of that work is to encapsulate the protocol behavior on well defined abstractions and to facilitate group formation of entities whom want to collaborate on a certain protocol. Our work does not deal with group formation, since we assume that service requests are sent to any device subscribed to the content of the service request. Our work focuses on facilitating the allocation of tasks between a number of mobile devices.

The development of mobile applications that leverage the cloud infrastructure is explored in [47]. [47] proposes a middleware capable of relocating specific parts of a application to be executed on the cloud, based on the quality criteria defined by the application developers. Our work also leverages from cloud computing, but we do not focus on optimizing the application execution. We focus on allowing the creation of collaborative applications on the mobile devices, leveraging the cloud as an infrastructure to interconnect the mobile devices.

7.4 Resource allocation on the cloud

Recently Cloud Computing has been used to improve the execution of applications on mobile devices regarding processing speed and power consumption.

Different aspects of the development of mobile applications using RESTful WS are described in [20]. The work of [61] targets battery optimization of battery by code offloading to the cloud. The work of [100] presents a web-services based Java classloader called (WSBCL). WSBCL is able to load Java classes stored on remote computers without the need of adding any other software servers to these computers.

Regarding resource allocation strategies we can cite [13], that proposes a decentralized approach to dynamically adapt cloud resource's usage. Resource allocation in utility computing is furthermore targeted in [91]. The resource allocation strategy presented in [91] pursues overall computation cost optimization.

Another approach to optimizing resource usage and guaranteeing SLA's is presented in [9]. [9] describes a game theoretical model to devise strategies for SaaS providers to schedule new machines on cloud providers. Our model focuses on minimizing the total time for completing a job on the cloud, while at the same time avoiding having too many unused resources.

Chapter 8

Conclusion

In this thesis we presented a pro-active approach, called Delegate MAS, to create service compositions that dynamically search and bind to component services. It allows the creation of composite services that can select component services based on the future QoS and which work in dynamic environments, where new services may become available, services may become unavailable, service quality may degrade, or services may even fail.

8.1 Summary

We created our approach based on the problems that we found with current approaches to dynamically bind composite services to their service counterparts, which are:

- Monitoring the current state of several services leads to a system that can at most simply react to changes after they have occurred. The system as a whole misses opportunities to better use the available resources.
- QoS information is independently gathered by different composite services. Independently monitoring the set of available services increases the load on the network and on the component services.
- Service's usage is not coordinated. Not coordinating service re-bindings under large scale failures can lead to chaotic situations, even leading to the complete halt of the system.

These problems lead to the research objective of creating a mechanism to allow dynamic rebinding of services that is decentralized, scalable, and robust to changes in the environment.

The thesis shows Delegate MAS, that spreads information on the network in order to coordinate the actions of independent composite services. A composite service has several tasks which are delegated to other services in the system. Delegate MAS takes advantage of knowing the graph structure of composite services to optimize the spreading of information to other services in the system. This information allows a better coordination with other services in the system that benefit from it, adjusting their own plans accordingly. A composite service, for instance, may change its plans of using a particular component service and decide to search for an alternative component service for a particular operation.

The service systems we studied are very large, having thousands of services interacting at any moment. Finding which service is the best one to be used at a particular time is challenging. We use ACO techniques to efficiently explore a very large service system and find suitable component services to participate in a service composition.

In order to have a realistic evaluation of our mechanism, we implemented a prototype that was deployed on a computer cluster. We used this prototype to perform several experiments with our mechanism.

We performed a thorough evaluation of our approach in different scenarios, from systems having a few hundred services to very large systems, having thousands of services. Besides evaluating our mechanism in systems of different sizes, in number of services, we performed experiments to show the behaviour of our mechanism under failure conditions.

We also created a software infrastructure to perform distributed systems experiments. This software infra-structure provides mechanisms to distribute the service deployment over a number of computers and to perform perform experiments. It facilitates sending experiments messages to the deployed services, and to shutdown services if needed.

8.2 Future Work

Although we created and tested a mechanism for service selection and composition, there is still a lot of future work to be done.

In our approach, each agent has a number of fixed parameters that they use to decide which actions to take, for instance, an exploration agent has to decide

when to explore new services, and when to simply use services that it already knows. Currently the parameters that define the exploration behaviour of such agents are fixed in order to avoid instability in the system. However, having fixed parameters also hinders the agent's ability to quickly adapt to unexpected changes in the environment. Hence, a future work is adding self-adaptive capabilities to the agents in our approach, for instance, allowing them to change their exploration parameters, or objective functions during run-time.

A challenge of adding self-adaptive capabilities to the agents in our approach, is that it becomes hard to predict the behaviour, and to avoid instability in the system. On the other hand, we foresee benefits, such as having a system that demands less manual tuning of parameters and less human intervention in case of changes in the environment where the system operates.

In our research we have studied how a Delegate MAS system works in two types of underlying network topologies, namely a random network and a internet based topology, which was created based on the internet topology data collected at the Skitter project [64]. While studying Delegate MAS in large networks, having 16k and 64k services, we noticed that small differences in the network parameters, such as the average vertex degree and the number of communities, had a great impact in the communication overhead of Delegate MAS.

A future work, related to the different topologies, is to further study how different network topologies affect the communication overhead of Delegate MAS and whether there is a significant impact in the reliability of service systems constructed using Delegate MAS. The knowledge about the network topology can, then, be added to the agents participating in the system so that will improve their performance in finding new services and quickly adapt to changes in the network.

We have explored how our approach reacts to large scale failures, and showed that it helps to create systems that are more resilient to failures, by using advanced reservations and dynamic rebinding of component services. An interesting future work is to extend our approach to accommodate the failures experienced by the system as input for adapting the agents behaviour so that they can learn with such failures and get even more robust to them. If the system can learn with past failures and get better with each new failure that happens, the system will be able to "survive" for a very long time.

Antifrafile systems not only survive crashes, or large scale disruption, but get better with them [103, 97]. Our approach provides initial steps towards creating antifragile systems, for instance, by avoiding single point of failures, and letting the system adapt when failures happen. What our current research does not provide is a systematic way for the system to learn with the previous system

disruptions. The information spread by the agents in Delegate MAS focus on the quality of previous service invocations. In an antifragile extension, the agents can spread information about previous system disruption, and each agent could, then, learn how to identify disruptions and how to behave when a disruption was detected.

8.3 Closing Reflection

Reflecting back on the evolution of the field of service selection and composition we see a different reality than what was predicted a few years ago. Organizations do not expose services to be automatically composed and orchestrated by third parties. The multi-organization aspect of service composition has, to a certain extent, diminished. Currently, the composition of services is done by using `API`'s from large service vendors.

Organizations have a growing number of micro-services, which are services that provide even more specialized functionality. Micro-services are being used for groups of developers so as to expose their software to other parts of the organization. Thus, the problem of service selection and composition continues to be relevant, but in a different way than predicted years ago.

We believe that more and more systems will be constructed by the composition of several services which are scattered over different datacenters or cloud providers around the world. Such systems will face constant pressure from the dynamic environments they operate and the constant problems such environments present. The scale of systems will also become a bigger concern and a more common issue with applications which serve millions of users every day. Constructing large scale applications, which operate by the composition of several services, will lead to the need of decentralized solutions that can autonomously cope with problems which arise in such systems.

However, we know it is hard for developers to give up control of their applications because developers expect to have strong guarantees about the execution of their applications. Our approach can not give strong deterministic guarantees about the execution of the services but it gives probabilistic guarantees which state that given a set number of constraints the approach will "normally" operate in the desired way.

We believe, though, that the concept of complete system control in very large service systems is actually unachievable. Very large service systems involve too many components which are prone to fail. Those failures vary from physical components, such as hard disks, processors, network cards, to

software components. Furthermore, failures can be added as bugs in the software, as bugs in the deployment procedures, in the service's configuration, etc. Having thousands of such physical and virtual components working together is challenging. A complete system control is thus impossible to achieve, since at best, it is possible to model the failures which affect the system and hope the model is close enough to reality. We believe that it is better to accept failures as first class citizens of very large systems and, as a consequence, to design such systems according to this acceptance.

Since it is not possible to foresee all possible ways a system can possibly fail, we believe it does not make sense to demand deterministic guarantees from any realistic system. Hence we believe approaches similar to ours, that embrace failure, and which allow applications to continue operating by coordinating their actions, are the best way to create large scale service systems, even knowing they still have a long way to improve.

Bibliography

[1] Agha, G. Actors: a model of concurrent computation in distributed systems. Tech. rep., MIT, 1985. pages 111

[2] Aguilera, M. K., Chen, W., and Toueg, S. Failure detection and consensus in the crash-recovery model. *Distributed Computing 13*, 2 (2000), 99–125. pages 34

[3] Albani, A., Keiblinger, A., Turowski, K., and Winnewisser, C. Identification and modelling of web services for inter-enterprise collaboration exemplified for the domain of strategic supply chain development. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, R. Meersman, Z. Tari, and D. Schmidt, Eds., vol. 2888 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003, pp. 74–92. pages 9

[4] Alonso, G., Casati, F., Kuno, H., and Machiraju, V. *Web Services Concepts, Architectures and Applications*. Springer, 2004. pages 10

[5] Alrifai, M., and Risse, T. Combining global optimization with local selection for efficient qos-aware service composition. In *Proceedings of the 18th international conference on World wide web* (New York, NY, USA, 2009), WWW '09, ACM, pp. 881–890. pages 117

[6] Alrifai, M., Skoutas, D., and Risse, T. Selecting skyline services for qos-based web service composition. In *Proceedings of the 19th international conference on World wide web* (New York, NY, USA, 2010), WWW '10, ACM, pp. 11–20. pages 117

[7] Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guizar, A., Kartha, N., et al. Web services business process execution language version 2.0. *OASIS Standard 11* (2007). pages 16

[8] Araragi, T., Takata, S., and Naoyuki, N. A verification method for a commitment strategy of the BDI architecture. *Computational Logic in Multi-Agent Systems* (2002), 109. pages 19

[9] Ardagna, D., Panicucci, B., and Passacantando, M. A game theoretic formulation of the service provisioning problem in cloud systems. In *Proceedings of the 20th international conference on World wide web* (New York, NY, USA, 2011), WWW '11, ACM, pp. 177–186. pages 119

[10] Baresi, L., and Guinea, S. Self-supervising bpel processes. *Software Engineering, IEEE Transactions on PP*, 99 (2010), 1 –1. pages 15

[11] Bazzan, A. A distributed approach for coordination of traffic signal agents. *Autonomous Agents and Multi-Agent Systems 10*, 1 (2005), 131–164. pages 99

[12] Bonabeau, E., Dorigo, M., and Theraulaz, G. *Swarm intelligence: from natural to artificial systems.* Oxford University Press, USA, 1999. pages 23

[13] Bonvin, N., Papaioannou, T., and Aberer, K. Autonomic SLA-Driven Provisioning for Cloud Applications. In *11th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid 2011)* (may 2011), pp. 434 –443. pages 119

[14] Braem, M., Verlaenen, K., Joncheere, N., Vanderperren, W., Van Der Straeten, R., Truyen, E., Joosen, W., and Jonckers, V. Isolating process-level concerns using padus. In *Business Process Management*, S. Dustdar, J. Fiadeiro, and A. Sheth, Eds., vol. 4102 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 113–128. pages 115

[15] Bratman, M. E. *Intention, plans, and practical reason.* Cambridge University Press, 1999. pages 17

[16] Buyens, K., Scandariato, R., and Joosen, W. Least privilege analysis in software architectures. *Software and System Modeling 12*, 2 (2013), 331–348. pages 38

[17] Castro, M., Costa, M., and Rowstron, A. Debunking some myths about structured and unstructured overlays. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2* (Berkeley, CA, USA, 2005), NSDI'05, USENIX Association, pp. 85–98. pages 26

[18] Charfi, A., Dinkelaker, T., and Mezini, M. A plug-in architecture for self-adaptive web service compositions. *Web Services, IEEE International Conference on 0* (2009), 35–42. pages 116

[19] Chechina, N., Trinder, P., Ghaffari, A., Green, R., Lundin, K., and Virding, R. The design of scalable distributed erlang. In *Proceedings of the Symposium on Implementation and Application of Functional Languages, Oxford, UK* (2012). pages 85

[20] Christensen, J. H. Using RESTful Web-Services and Cloud Computing to Create Next Generation Mobile Applications. In *Proc. of the 24th Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA '09)* (2009), ACM, pp. 627–634. pages 119

[21] Claes, R., and Holvoet, T. Weighing communication overhead against travel time reduction in advanced traffic information systems. *Progress in Artificial Intelligence 1*, 2 (July 2012), 165–172. pages 39

[22] Claes, R., and Holvoet, T. Traffic coordination using aggregation-based traffic predictions. *IEEE Intelligent Systems 29*, 4 (July 2014), 96–100. pages 113

[23] Claes, R., Holvoet, T., and Weyns, D. A decentralized approach for anticipatory vehicle routing using delegate multiagent systems. *IEEE Transactions on Intelligent Transportation Systems 12*, 2 (March 2011), 364–373. pages 113

[24] Clair, G., Kaddoum, E., Gleizes, M.-P., and Picard, G. Self-Regulation in Self-Organising Multi-Agent Systems for Adaptive and Intelligent Manufacturing Control. In *IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)* (2008), pp. 107–116. pages 117

[25] Clark, D., Lehr, B., Bauer, S., Faratin, P., Sami, R., and Wroclawski, J. Overlay networks and the future of the internet. *Communications and Strategies 63* (2006), 109. pages 26

[26] Colombo, M., Nitto, E. D., and Mauri, M. SCENE: A service composition execution environment supporting dynamic changes disciplined through rules. In *Service-Oriented Computing â€“ ICSOC 2006* (2006), vol. 4294 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 191–202. pages 16, 115

[27] Costa, P., Napper, J., Pierre, G., and van Steen, M. Autonomous resource selection for decentralized utility computing. In *Distributed Computing Systems, 2009. ICDCS '09. 29th IEEE International Conference on* (2009), pp. 561 –570. pages 60

[28] Cristian, F., and Fetzer, C. The timed asynchronous distributed system model. *Parallel and Distributed Systems, IEEE Transactions on 10*, 6 (1999), 642–657. pages 36

[29] Cruz Torres, M. H., Haesevoets, R., and Holvoet, T. CooS: coordination support for mobile collaborative applications. In *Mobiquitous - International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, Beijing, China, 12-14 December 2012* (2012). Accepted. pages 84

[30] Cruz Torres, M. H., Noël, V., Holvoet, T., and Arcangeli, J.-P. Mas organisations to adapt your composite service. In *Proceedings of the 3rd International Workshop on Monitoring, Adaptation and Beyond* (New York, NY, USA, 2010), MONA '10, ACM, pp. 33–39. pages 83

[31] DeLoach, S. A. *OMACS: A Framework for Adaptive, Complex Systems*. Information Science Reference, 2008, ch. IV. pages 88

[32] Deugo, D., Weiss, M., and Kendall, E. Reusable patterns for agent coordination. In *in: Omicini, A., Coordination of Internet Agents*, Springer, pp. 347–368. pages 21

[33] Di Nitto, E., Ghezzi, C., Metzger, A., Papazoglou, M., and Pohl, K. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering 15*, 3-4 (Dec. 2008), 313–341. pages 14

[34] Dignum, V., Ed. *The Role of Organization in Agent Systems*. Information Science Reference, 2008, ch. I, pp. 1–17. pages 88

[35] Dorigo, M., Caro, G. D., and Gambardella, L. M. Ant algorithms for discrete optimization. *Artificial Life 5* (1999), 137–172. pages 24

[36] Dorigo, M., and Stützle, T. The ant colony optimization metaheuristic: Algorithms, applications, and advances. In *Handbook of metaheuristics*. Springer, 2003, pp. 250–285. pages 24

[37] Driscoll, K., Hall, B., Sivencrona, H., and Zumsteg, P. Byzantine fault tolerance, from theory to reality. In *Computer Safety, Reliability, and Security*. Springer, 2003, pp. 235–248. pages 34

[38] Ducatelle, F., Di Caro, G. A., and Gambardella, L. M. Principles and applications of swarm intelligence for adaptive routing in telecommunications networks. *Swarm Intelligence 4*, 3 (2010), 173–198. pages 23

[39] Dustdar, S., and Schreiner, W. A survey on web services composition. *International Journal of Web and Grid Services 1*, 1 (2005), 1–30. pages 39

[40] Eugster, P. T., Garbinato, B., and Holzer, A. Location-based publish/subscribe. In *Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications* (Washington, DC, USA, 2005), NCA '05, IEEE Computer Society, pp. 279–282. pages 104, 105

[41] Ezenwoye, O., and Sadjadi, S. M. A proxy-based approach to enhancing the autonomic behavior in composite services. *JNW 3*, 5 (2008), 42–53. pages 115

[42] Famaey, J., Wauters, T., De Turck, F., Dhoedt, B., and Demeester, P. Towards efficient service placement and server selection for large-scale deployments. In *Telecommunications, 2008. AICT '08. Fourth Advanced International Conference on* (june 2008), pp. 13 –18. pages 60

[43] Fette, I., and Melnikov, A. The WebSocket Protocol. RFC 6455 (Proposed Standard), Dec. 2011. pages 108

[44] Fielding, R. T. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000. pages 13

[45] Fitoussi, D., and Tennenholtz, M. Choosing social laws for multi-agent systems: Minimality and simplicity. *Artificial Intelligence 119*, 1 (2000), 61–101. pages 20

[46] Ghezzi, C., Motta, A., Panzica La Manna, V., and Tamburrelli, G. Qos driven dynamic binding in-the-many. In *Research into Practice – Reality and Gaps*, G. Heineman, J. Kofron, and F. Plasil, Eds., vol. 6093 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010, pp. 68–83. pages 117

[47] Giurgiu, I., Riva, O., Juric, D., Krivulev, I., and Alonso, G. Calling the cloud: Enabling mobile phones as interfaces to cloud applications. In *Middleware 2009*, J. Bacon and B. Cooper, Eds., vol. 5896 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2009, pp. 83–102. pages 118, 119

[48] Gudgin, M., Hadley, M., and Rogers, T. Web services addressing 1.0 - core. World Wide Web Consortium, Recommendation REC-ws-addr-core-20060509, May 2006. pages 88

[49] Gutierrez-Garcia, J. O. Agent-based cloud workflow execution. *Integrated Computer-Aided Engineering Volume 19* (2012), 39 Last Page – 56. pages 22

[50] Haesevoets, R., Weyns, D., Holvoet, T., and Joosen, W. A formal model for self-adaptive and self-healing organizations. 116–125. pages 20

[51] Hoffert, J., Schmidt, D. C., and Gokhale, A. S. Adapting distributed real-time and embedded pub/sub middleware for cloud computing environments. In *Middleware* (2010), pp. 21–41. pages 118

[52] Hohpe, G., and Woolf, B. *Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, October 2003. pages 115

[53] Holvoet, T., Weyns, D., and Valckenaers, P. Patterns of delegate mas. *Self-Adaptive and Self-Organizing Systems, International Conference on 0* (2009), 1–9. pages 41

[54] Hull, R., and Su, J. Tools for composite web services: A short overview. *SIGMOD Rec. 34*, 2 (June 2005), 86–95. pages 15

[55] Jelasity, M., Montresor, A., and Babaoglu, O. T-man: Gossip-based fast overlay topology construction. *Computer Networks 53*, 13 (2009), 2321 – 2339. <ce:title>Gossiping in Distributed Systems</ce:title>. pages 28

[56] Jennings, N. R. Coordination techniques for distributed artificial intelligence. *Foundations of distributed artificial intelligence* (1996), 187–210. pages 22

[57] Kermarrec, A., and Van Steen, M. Gossiping in distributed systems. *ACM SIGOPS Operating Systems Review 41*, 5 (2007), 2–7. pages 27

[58] Kinny, D., and Georgeff, M. *Commitment and effectiveness of situated agents*. Dept. of Computer Science, University of Melbourne, 1991. pages 18

[59] Koźlak, J., Créput, J.-C., Hilaire, V., and Koukam, A. Multi-agent approach to dynamic pick-up and delivery problem with uncertain knowledge about future transport demands. *Fundam. Inf. 71*, 1 (Jan. 2006), 27–36. pages 99

[60] Kshemkalyani, A. D., and Singhal, M. *Distributed computing: principles, algorithms, and systems*. Cambridge University Press, 2008. pages 36

[61] Kumar, K., and Lu, Y.-H. Cloud computing for mobile users: Can offloading computation save energy? *Computer 43*, 4 (april 2010), 51 –56. pages 119

[62] Kutanoglu, E., and Wu, S. On combinatorial auction and lagrangean relaxation for distributed resource scheduling. *IIE transactions 31*, 9 (1999), 813–826. pages 99

[63] Leitner, P., Michlmayr, A., Rosenberg, F., and Dustdar, S. Monitoring, prediction and prevention of sla violations in composite services. In *2010 IEEE International Conference on Web Services* (2010). pages 116

[64] Leskovec, J., Kleinberg, J., and Faloutsos, C. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining* (2005), ACM, pp. 177–187. pages 62, 123

[65] Luqman, F., and Griss, M. Overseer: a mobile context-aware collaboration and task management system for disaster response. In *The Eighth International Conference on Creating, Connecting and Collaborating through Computing, UC San Diego, La Jolla CA, United States* (2010). pages 99

[66] Mabrouk, N. B., Beauche, S., Kuznetsova, E., Georgantas, N., and Issarny, V. QoS-Aware service composition in dynamic service oriented environments. In *Middleware 2009* (2009), vol. 5896 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 123–142. pages 117

[67] Marconi, A., Pistore, M., Sirbu, A., Eberle, H., Leymann, F., and Unger, T. Enabling adaptation of pervasive flows: Built-in contextual adaptation. In *Service-Oriented Computing* (2009), vol. 5900 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 445–454. pages 116

[68] Matos, M., Sousa, A., Pereira, J., Oliveira, R., Deliot, E., and Murray, P. Clon: Overlay networks and gossip protocols for cloud environments. 549–566. pages 28

[69] Michlmayr, A., Rosenberg, F., Leitner, P., and Dustdar, S. End-to-end support for qos-aware service selection, binding and mediation in vresco. *Services Computing, IEEE Transactions on PP*, 99 (2010), 1 –1. pages 117

[70] Mosincat, A. D., and Binder, W. Self-tuning bpel processes. In *Proceedings of the 6th international conference on Autonomic computing* (New York, NY, USA, 2009), ICAC '09, ACM, pp. 47–48. pages 15

[71] Noël, V. *Component-based Software Architectures and Multi-Agent Systems: Mutual and Complementary Contributions for Supporting Software Development*. PhD thesis, 2012. pages 16

[72] Noël, V., Arcangeli, J.-P., and Gleizes, M.-P. Between Design and Implementation of Multi-Agent Systems: A Component-Based Two-Step Process. In *EUMAS'10* (2010). pages 93

[73] Nwana, H., Lee, L., and Jennings, N. Co-ordination in software agent systems. *To appear in: BT Technology Journal 14* (1996), 4. pages 19, 21

[74] Orgaz, G. B., Barrero, D. F., R-Moreno, M. D., and Camacho, D. Acquisition of business intelligence from human experience in route planning. *Enterprise Information Systems*, ahead-of-print (2013), 1–21. pages 114

[75] Papadopouli, M., and Schulzrinne, H. Connection sharing in an ad hoc wireless network among collaborating hosts. In *Proc. International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)* (1999), pp. 169–185. pages 99

[76] Papazoglou, M. P. *Web Services: Principles and Technology*. Pearson, Prentice Hall, 2008. pages 114

[77] Papazoglou, M. P., Traverso, P., Dustdar, S., and Leymann, F. Service-oriented computing: State of the art and research challenges. *Computer 40*, 11 (2007), 38–45. pages 86

[78] Parragh, S. N., Doerner, K. F., and Hartl, R. F. Variable neighborhood search for the dial-a-ride problem. *Computers & Operations Research 37*, 6 (2010), 1129 – 1138. pages 101

[79] Pautasso, C. Composing restful services with jopera. In *SC '09: Proceedings of the 8th International Conference on Software Composition* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 142–159. pages 16

[80] Pautasso, C., Zimmermann, O., and Leymann, F. Restful web services vs. "big"' web services: making the right architectural decision. In *Proceeding of the 17th international conference on World Wide Web* (New York, NY, USA, 2008), WWW '08, ACM, pp. 805–814. pages 11, 13

[81] Pereira, J. V. The new supply chain's frontier: Information management. *International Journal of Information Management 29*, 5 (2009), 372 – 379. pages 85

[82] Rao, A. S., and Georgeff, M. P. Bdi agents: From theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)* (Cambridge, MA, USA, June 1995), L. Victor and G. Les, Eds., MIT Press, pp. 312–319. pages 17

[83] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev. 31*, 4 (Aug. 2001), 161–172. pages 26

[84] Reid, C. R., Sumpter, D. J. T., and Beekman, M. Optimisation in a natural system: Argentine ants solve the towers of hanoi. *J Exp Biol 214*, 1 (2011), 50–58. pages 24

[85] Rocha, R., Cunha, A., Varandas, J., and Dias, J. Towards a new mobility concept for cities: architecture and programming of semi-autonomous electric vehicles. *Industrial Robot: An International Journal 34*, 2 (2007), 142–149. pages 99

[86] Rosenschein, J. S. *Rules of encounter: designing conventions for automated negotiation among computers*. MIT press, 1994. pages 39

[87] Sadeh, N., Hildum, D., Kjenstad, D., and Tseng, A. Mascot: an agent-based architecture for dynamic supply chain creation and coordination in the internet economy. *Production Planning & Control 12*, 3 (2001), 212–223. pages 99

[88] Schelfthout, K., Weyns, D., and Holvoet, T. Middleware for protocol-based coordination in mobile applications. *IEEE Distributed Systems Online 7*, 8 (August 2006), 1–18. pages 118, 119

[89] Schmidt, D. C. Middleware for real-timeand embedded systems. Communications of the ACM, 2002. pages 100, 118

[90] Shoham, Y., and Tennenholtz, M. On social laws for artificial agent societies: off-line design. *Artificial Intelligence 73*, 1–2 (1995), 231 – 252. Computational Research on Interaction and Agency, Part 2. pages 20

[91] Silva, J. a. N., Veiga, L., and Ferreira, P. Heuristic for Resources Allocation on Utility Computing Infrastructures. In *Proc. of the 6th Int. Workshop on Middleware for Grid Computing (MGC 2008)* (2008), ACM, pp. 1–6. pages 119

[92] Skeen, D., and Stonebraker, M. A formal model of crash recovery in a distributed system. *Software Engineering, IEEE Transactions on SE-9*, 3 (May 1983), 219–228. pages 34

[93] Specification, F. Link: http://www. fipa. org/specs/fipa00029. *SC00029H. html* (2003). pages 21, 99

[94] Stein, S., Payne, T., and Jennings, N. Robust execution of service workflows using redundancy and advance reservations. *Services Computing, IEEE Transactions on 4*, 2 (feb. 2011), 125 –139. pages 22, 117, 118

[95] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev. 31* (August 2001), 149–160. pages 26

[96] Strunk, A., Braun, I., Reichert, S., and Schill, A. Supporting rebinding in bpel. In *Web Services, 2009. ICWS 2009. IEEE International Conference on* (july 2009), pp. 864 –871. pages 114

[97] Taleb, N. N. *Antifragile: how to live in a world we don't understand.* Allen Lane, 2012. pages 123

[98] Tanenbaum, A. S., and "van" Steen, M. *Distributed systems*, vol. 2. Prentice Hall Upper Saddle River, 2002. pages 36

[99] Tarkoma, S. *Overlay Networks: Toward Information Networking.*, 1st ed. Auerbach Publications, Boston, MA, USA, 2010. pages 26

[100] Teixeira, F., Santana, M., Santana, R., Bruschi, S., and Estrella, J. WSBCL: Web Services Based Classloader. In *20th IEEE Int. Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2011)* (2011), IEEE, pp. 128–133. pages 119

[101] Thangarajah, J., Padgham, L., and Harland, J. Representation and reasoning for goals in BDI agents. *Australian Computer Science Communications 24*, 1 (2002), 259–265. pages 17

[102] Theraulaz, G., and Bonabeau, E. A brief history of stigmergy. *Artificial life 5*, 2 (1999), 97–116. pages 23

[103] Tseitlin, A. The antifragile organization. *Communications of the ACM 56*, 8 (2013), 40–44. pages 123

[104] Ueyama, J., Pinto, V. P. V., Madeira, E. R. M., Grace, P., Jonhson, T. M. M., and Camargo, R. Y. Exploiting a generic approach for constructing mobile device applications. COMSWARE '09, ACM, pp. 12:1–12:12. pages 118

[105] Van Brussel, H. Holonic manufacturing systems. In *CIRP Encyclopedia of Production Engineering*. Springer, 2014, pp. 654–659. pages 114

[106] Vandael, S., Holvoet, T., and Deconinck, G. Aggregate demand models for electric vehicles in a smart electricity grid. In *Proceedings of the Fourth International Workshop on Agent Technologies for Energy Systems (ATES 2013)* (May 2013), pp. 1–2. pages 39

[107] Vargha, A., and Delaney, H. D. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics 25*, 2 (2000), 101–132. pages 75

[108] Verstraete, P., Saint Germain, B., Valckenaers, P., Van Brussel, H., Belle, J., and Hadeli, H. Engineering manufacturing control systems using prosa and delegate mas. *International Journal of Agent-Oriented Software Engineering 2*, 1 (2008), 62–89. pages 114

[109] Vogels, W. Eventually consistent. *Commun. ACM 52*, 1 (Jan. 2009), 40–44. pages 37

[110] Voulgaris, S., Gavidia, D., and Van Steen, M. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management 13*, 2 (2005), 197–217. pages 28

[111] Walsh, W., and Wellman, M. A market protocol for decentralized task allocation. In *Multi Agent Systems, 1998. Proceedings. International Conference on* (1998), IEEE, pp. 325–332. pages 21

[112] Weiser, M. Some computer science issues in ubiquitous computing. *Communications of the ACM 36*, 7 (1993), 75–84. pages 99

[113] Weiss, G., Ed. *Multiagent Systems (Intelligent Robotics and Autonomous Agents series)*. The MIT Press; second edition edition, 2013. pages 19

[114] Weyns, D., Boucké, N., Holvoet, T., and Demarsin, B. Dyncnet: A protocol for dynamic task assignment in multiagent systems. *SASO 7* (2007), 281–284. pages 22

[115] Weyns, D., Boucké, N., Holvoet, T., and Demarsin, B. Dyncnet: A protocol for flexible transport assignment in agv transportation systems. CW Reports CW478, K.U.Leuven, Department of Computer Science, February 2007. pages 39

[116] Weyns, D., Haesevoets, R., Helleboogh, A., and Holvoet. The macodo organization model for context-driven dynamic agent organzations. *ACM Trans. Auton. Adapt. Syst. 6*, 4 (2010). pages xvi, 84, 88, 89

[117] Weyns, D., Haesevoets, R., Helleboogh, A., Holvoet, T., and Joosen, W. The macodo middleware for context-driven dynamic agent organizations. *TAAS, ACM Transactions on Autonomous and Adaptive Systems 5*, 1 (2010). pages 20

[118] Weyns, D., and Holvoet, T. Decentralized control of automatic guided vehicles applying multi-agent systems in practice, 2008. pages 40

[119] Weyns, D., Holvoet, T., and Schelfthout, K. Multiagent systems as software architecture: another perspective on software engineering with multiagent systems. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems* (2006), pp. 1314–1317. pages 16

[120] Wooldridge, M. *An introduction to multiagent systems*. Wiley. com, 2008. pages 16

[121] Zeldovich, N., Boyd-Wickizer, S., and Mazieres, D. Securing distributed systems with information flow control. In *NSDI* (2008), vol. 8, pp. 293–308. pages 38

[122] Zeng, L., Benatallah, B., H.H. Ngu, A., Dumas, M., Kalagnanam, J., and Chang, H. Qos-aware middleware for web services composition. *IEEE Trans. Softw. Eng. 30*, 5 (2004), 311–327. pages 117

List of Publications

Papers at International Conferences and Symposia, Published in Full in Proceedings

- Cruz Torres, Mario Henrique; Holvoet, Tom. **Self-adaptive resilient service composition**. In: Proceedings of the IEEE International Conference on Cloud and Autonomic Computing (ICCAC 2014), London, United Kingdom, pp. 141-150
- Cruz Torres, Mario Henrique; Haesevoets, Robrecht; Holvoet, Tom. **CooS: coordination support for mobile collaborative applications**. In: Proceedings of the 9th International Conference on Mobile and Ubiquitous Systems:Computing, Networking and Services (MobiQuitous 2012), Beijing, China, pp. 152-163
- Ferber, Marvin; Rauber, Thomas; Cruz Torres, Mario Henrique; Holvoet, Tom. **Resource allocation for cloud-assisted mobile applications**. In: Proceedings of the IEEE Fifth International Conference on Cloud Computing (Cloud 2012), Honolulu, USA, pp. 400-407
- Cruz Torres, Mario Henrique; Van Beers, Tony; Holvoet, Tom. **(No) More design patterns for multi-agent systems**. In: Proceedings of the compilation of the co-located workshops, DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, and VMIL'11 (Agere! 2011), Portland, USA, pp. 213-220
- Cruz Torres, Mario Henrique; Holvoet, Tom. **Towards robust service workflows: a decentralized approach**. In: Proceedings of the Cooperative Information Systems (CoopIS 2011), Crete, Greece, pp. 155-162
- Cruz Torres, Mario Henrique; Holvoet, Tom. **Composite service adaptation: a QoS-driven approach**. In: Proceedings of 5th

International Conference on COMmunication System SoftWAre and MiddlewaRE (COMSWARE 2011), Verona, Italy

- Haesevoets, Robrecht; Weyns, Danny; Cruz Torres, Mario Henrique; Helleboogh, Alexander; Holvoet, Tom; Joosen, Wouter. **A middleware model in Alloy for supply chain-wide agent interactions**. In: Proceedings of the 11th International Workshop on Agent Oriented Software Engineering (AOSE 2010), Toronto, Canada, pp. 189-204

- Cruz Torres, Mario Henrique; Noel, Victor; Holvoet, Tom; Arcangeli, Jean-Paul. **MAS organisations to adapt your composite service**. In: Proceedings of the 3rd International Workshop on Monitoring, Adaptation and Beyond (MONA+ 2010), Ayia Napa, Cyprus, pp. 33-39

Peer-Reviewed International Demos and Posters published in proceedings

- Cruz Torres, Mario Henrique; Holvoet, Tom. **Composite service optimization through decentralized coordination**. In: Proceedings of the 7th International conference on Autonomic computing (ICAC 2010), Karlsruhe, Germany, pp. 167-168

- Van Gompel, Jelle; Tuts, Bart; Claes, Rutger; Cruz Torres, Mario Henrique; Holvoet, Tom. **MAS-DisCoSim 4 PDP: A testbed for multi-agent solutions to PDPs**. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010), Toronto, Canada, pp. 1639-1640

FACULTY OF ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE
IMINDS-DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Heverlee
MarioHenrique.CruzTorres@cs.kuleuven.be
http://www.cs.kuleuven.be